# **[引言](http://www.cnblogs.com/wayfarer/articles/241012.html)**

软件设计因为引入面向对象思想而逐渐变得丰富起来。“一切皆为对象”的精义，使得程序世界所要处理的逻辑简化，开发者可以用一组对象以及这些对象之间的关系将软件系统形象地表示出来。而从对象的定义，进而到模块，到组件的定义，利用面向对象思想的封装、继承、多态的思想，使得软件系统开发可以向搭建房屋那样，循序渐进，从砖石到楼层，进而到整幢大厦的建成。应用面向对象思想，在设计规模更大、逻辑更复杂的系统时，开发周期反而能变的更短。自然其中，需要应用到软件工程的开发定义、流程的过程控制，乃至于质量的缺陷管理。但从技术的细节来看，面向对象设计技术居功至伟。然而，面向对象设计的唯一问题是，它本质是静态的，封闭的，任何需求的细微变化都可能对开发进度造成重大影响。

可能解决该问题的方法是设计模式。GOF将面向对象软件的设计经验作为设计模式纪录下来，它使人们可以更加简单方便地复用成功的设计和体系结构，帮助开发人员做出有利于系统复用的选择。设计模式解决特定的设计问题，使面向对象设计更灵活、优雅，最终复用性更好。然而，设计模式虽然给了我们设计的典范与准则，通过最大程度的利用面向对象的特性，诸如利用继承、多态，对责任进行分离、对依赖进行倒置，面向抽象，面向接口，最终设计出灵活、可扩展、可重用的类库、组件，乃至于整个系统的架构。在设计的过程中，通过各种模式体现了对象的行为，暴露的接口，对象间关系，以及对象分别在不同层次中表现出来的形态。然而鉴于对象封装的特殊性，“设计模式”的触角始终在接口与抽象中大做文章，而对于对象内部则无能为力。

Aspect-Oriented Programming（面向方面编程，AOP）正好可以解决这一问题。它允许开发者动态地修改静态的OO模型，构造出一个能够不断增长以满足新增需求的系统，就象现实世界中的对象会在其生命周期中不断改变自身，应用程序也可以在发展中拥有新的功能。AOP利用一种称为“横切”的技术，剖解开封装的对象内部，并将那些影响了多个类的行为封装到一个可重用模块，并将其名为“Aspect”，即方面。所谓“方面”，简单地说，就是将那些与业务无关，却为业务模块所共同调用的逻辑或责任，例如事务处理、日志管理、权限控制等，封装起来，便于减少系统的重复代码，降低模块间的耦合度，并有利于未来的可操作性和可维护性。

面向方面编程（AOP）是施乐公司帕洛阿尔托研究中心（Xerox PARC）在上世纪90年代发明的一种编程范式。但真正的发展却兴起于近几年对软件设计方兴未艾的研究。由于软件系统越来越复杂，大型的企业级应用越来越需要人们将核心业务与公共业务分离。AOP技术正是通过编写横切关注点的代码，即“方面”，分离出通用的服务以形成统一的功能架构。它能够将应用程序中的商业逻辑同对其提供支持的通用服务进行分离，使得开发人员从重复解决通用服务的劳动中解脱出来，而仅专注于企业的核心商业逻辑。因此，AOP技术也就受到越来越多的关注，而应用于各种平台下的AOP技术也应运而生。但由于AOP技术相对于成熟的OOP技术而言，在性能、稳定性、适用性等方面还有待完善，同时AOP技术也没有形成一个统一的标准，这使得AOP技术的研究更具有前沿性的探索价值。

<http://wayfarer.cnblogs.com/articles/241012.html>

# **[AOP技术基础](http://www.cnblogs.com/wayfarer/articles/241024.html)**

2.1 AOP技术起源

AOP技术的诞生并不算晚，早在1990年开始，来自Xerox Palo Alto Research Lab（即PARC）的研究人员就对面向对象思想的局限性进行了分析。他们研究出了一种新的编程思想，借助这一思想或许可以通过减少代码重复模块从而帮助开发人员提高工作效率。随着研究的逐渐深入，AOP也逐渐发展成一套完整的程序设计思想，各种应用AOP的技术也应运而生。

AOP技术在Java平台下是最先得到应用的。就在PARC对于面向方面编程进行研究的同时，美国Northeastern University的博士生Cristina Lopes和其同事也开始了类似的思考。最终，美国国防先进技术研究计划署（Defense Advanced Research Projects Agency即DARPA）注意到了这项工作，并提供了科研经费，鼓励将二者的工作成果结合起来。他们通过定义一套Java语言的扩展系统，使开发者可以方便的进行面向方面的开发，这套扩展系统被称为AspectJ。之后，AspectJ在2002年被转让给Eclipse Foundation，从而成为在开源社区中AOP技术的先锋，也是目前最为流行的AOP工具。

AspectWerkz则是基于Java的动态的、轻量级AOP框架。AspectWerkz仍然是开源社区中的产品，由BEA System提供赞助，开发者则是BEA的两名员工Jonas Bonér和Alexandre Vasseur。最近版本是AspectWerkz 2.0。2005年1月，AspectJ和AspectWerkz达成协议，同意将二者的成果综合到一起，取其精华创建一个单一的工具。他们合作的第一个发布版本为AspectJ 5，它扩展了AspectJ语言，以支持基于Annotation开发风格而又支持类似AspectJ代码风格。AspectJ 5也为Java 5的语言特性提供完全的AOP支持。

在Java阵营中，商用软件制造商JBoss在其2004年推出的JBoss 4.0中，引入了AOP框架和组件。在JBoss 4.0中，用户可以在JBoss应用服务器外部单独使用JBoss AOP，该版本为JBoss AOP 1.0，是在2004年10月发布的。在2005年，JBoss AOP框架又发布了1.3.0版本，新版本对加载期织入（Weev）和切点（point cut）匹配的性能做了很大的优化，使应用程序的启动时间大大缩短。

作为轻型的Framework，Spring在开发轻量级的J2EE时，应用是非常广泛的。它通过IoC模式（Inversion of Control，控制反转模式）来实现AOP，通常被称为Spring AOP。在2004年，被作为Spring框架的扩展而发布，目前版本已更新到1.1.3。Spring AOP作为一种非侵略性的，轻型的AOP框架，开发者无需使用预编译器或其他的元标签，在Java程序中应用AOP。目前，AOP的功能完全集成到了Spring事务管理、日志和其他各种特性的上下文中。

在.Net的阵营中，AOP技术的应用远不如Java阵营对AOP的关注。2005年1月，微软发布的Enterprise Library提供了7种不同的“应用程序块（application blocks）”。有个别专家认为，这些组件可以被认为是方面。但该观点并未得到一致的认同。事实上，在.Net平台下，推动AOP技术发展的原动力并非微软，而是开源社区。虽然，微软的技术专家们亦然听到了在.Net Framework中增加AOP技术的群众呼声，但作为如此巨大的软件公司，要让它灵活地转变战略方向，显然是不太现实的。正因为此，才赐予了开源社区在AOP技术的研究与探索上一个巨大的发展空间。

与Java阵营中的AOP技术不同，目前在.Net平台下的各种AOP工具，基本上还停留在实验室阶段。但一些在技术上领先且逐渐成熟的AOP产品，也在开源社区中渐露峥嵘。这其中主要包括Aspect#，AspectDNG，Eos AOP等。

Aspect#是基于Castle动态代理技术来实现的。Castle源于Apache Avalon项目，其目的在于实现一个轻量级的IoC容器。Aspect#于2005年6月被收录为Castle的其中一个子项目。它是针对CLI（.Net和Mono）实现的AOP框架，利用了反射、代理等机制。目前的Aspect#版本为2.1.1。

AspectDNG目前的版本为0.7，仍然处于beta版的阶段。它的实现技术是基于rail的静态织入。Rail属于IL级别下的代码织入，它自定义的一套xml格式的ILML语言，能够将原有的程序集拆散成ILML格式，以便于对静态程序集进行修改和扩展，从而达到静态织入的目的。因为AspectDNG是属于IL级别下的代码织入，因此在.Net平台下，并不受具体的编程语言限制。

Eos AOP与AspectDNG一样，仍然采用静态织入的方式，但从语法定义上，它更近似于AspectJ关于AOP的实现。它扩展了C#语法，引入了aspect、introduce、before、after等关键字，并且提供了专用的Eos编译器。Eos项目是于2004年9月开始启动，2005年6月推出的0.3.3版本为最新版本，主要的开发人员为Hridesh Rajan和Kevin Sullivan。前者为Virginia大学计算机系的研究生，Eos项目最初是由Hridesh Rajan提出的；而后者则为该计算机系的副教授（Associate Professor）。所以自Eos诞生之初，就带有浓厚的学院派特色。

从AOP技术的整体发展来看，高性能、稳定、可扩展、易用的AOP框架是其趋势与目标。从上述对各种AOP技术的分析来看，AOP技术无疑是具有共同特点的，而各种实现技术就是围绕着这些共性深入与延伸。接下来，我将概要地介绍AOP的本质，以及它的技术要素。

2.2 AOP技术本质

2.2.1 技术概览

AOP（Aspect-Oriented Programming，面向方面编程），可以说是OOP（Object-Oriented Programing，面向对象编程）的补充和完善。OOP引入封装、继承和多态性等概念来建立一种对象层次结构，用以模拟公共行为的一个集合。当我们需要为分散的对象引入公共行为的时候，OOP则显得无能为力。也就是说，OOP允许你定义从上到下的关系，但并不适合定义从左到右的关系。例如日志功能。日志代码往往水平地散布在所有对象层次中，而与它所散布到的对象的核心功能毫无关系。对于其他类型的代码，如安全性、异常处理和透明的持续性也是如此。这种散布在各处的无关的代码被称为横切（cross-cutting）代码，在OOP设计中，它导致了大量代码的重复，而不利于各个模块的重用。

而AOP技术则恰恰相反，它利用一种称为“横切”的技术，剖解开封装的对象内部，并将那些影响了多个类的公共行为封装到一个可重用模块，并将其名为“Aspect”，即方面。所谓“方面”，简单地说，就是将那些与业务无关，却为业务模块所共同调用的逻辑或责任封装起来，便于减少系统的重复代码，降低模块间的耦合度，并有利于未来的可操作性和可维护性。AOP代表的是一个横向的关系，如果说“对象”是一个空心的圆柱体，其中封装的是对象的属性和行为；那么面向方面编程的方法，就仿佛一把利刃，将这些空心圆柱体剖开，以获得其内部的消息。而剖开的切面，也就是所谓的“方面”了。然后它又以巧夺天功的妙手将这些剖开的切面复原，不留痕迹。

使用“横切”技术，AOP把软件系统分为两个部分：核心关注点和横切关注点。业务处理的主要流程是核心关注点，与之关系不大的部分是横切关注点。横切关注点的一个特点是，他们经常发生在核心关注点的多处，而各处都基本相似。比如权限认证、日志、事务处理。Aop 的作用在于分离系统中的各种关注点，将核心关注点和横切关注点分离开来。正如Avanade公司的高级方案构架师Adam Magee所说，AOP的核心思想就是“将应用程序中的商业逻辑同对其提供支持的通用服务进行分离。”

实现AOP的技术，主要分为两大类：一是采用动态代理技术，利用截取消息的方式，对该消息进行装饰，以取代原有对象行为的执行；二是采用静态织入的方式，引入特定的语法创建“方面”，从而使得编译器可以在编译期间织入有关“方面”的代码。然而殊途同归，实现AOP的技术特性却是相同的，分别为：

1、join point（连接点）：是程序执行中的一个精确执行点，例如类中的一个方法。它是一个抽象的概念，在实现AOP时，并不需要去定义一个join point。  
2、point cut（切入点）：本质上是一个捕获连接点的结构。在AOP中，可以定义一个point cut，来捕获相关方法的调用。  
3、advice（通知）：是point cut的执行代码，是执行“方面”的具体逻辑。  
4、aspect（方面）：point cut和advice结合起来就是aspect，它类似于OOP中定义的一个类，但它代表的更多是对象间横向的关系。  
5、introduce（引入）：为对象引入附加的方法或属性，从而达到修改对象结构的目的。有的AOP工具又将其称为mixin。

上述的技术特性组成了基本的AOP技术，大多数AOP工具均实现了这些技术。它们也可以是研究AOP技术的基本术语。

2.2.2 横切技术

“横切”是AOP的专有名词。它是一种蕴含强大力量的相对简单的设计和编程技术，尤其是用于建立松散耦合的、可扩展的企业系统时。横切技术可以使得AOP在一个给定的编程模型中穿越既定的职责部分（比如日志记录和性能优化）的操作。

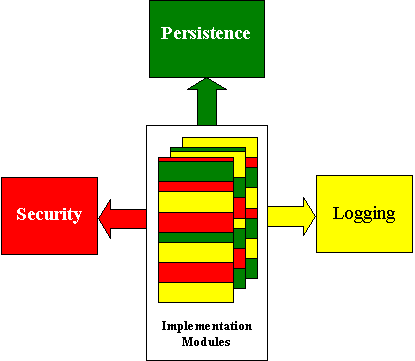
如果不使用横切技术，软件开发是怎样的情形呢？在传统的程序中，由于横切行为的实现是分散的，开发人员很难对这些行为进行逻辑上的实现或更改。例如，用于日志记录的代码和主要用于其它职责的代码缠绕在一起。根据所解决的问题的复杂程度和作用域的不同，所引起的混乱可大可小。更改一个应用程序的日志记录策略可能涉及数百次编辑——即使可行，这也是个令人头疼的任务。

在AOP中，我们将这些具有公共逻辑的，与其他模块的核心逻辑纠缠在一起的行为称为“横切关注点（Crosscutting Concern）”，因为它跨越了给定编程模型中的典型职责界限。

2.2.2.1 横切关注点

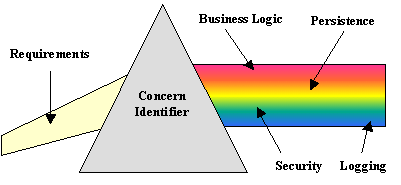
一个关注点（concern）就是一个特定的目的，一块我们感兴趣的区域，一段我们需要的逻辑行为。从技术的角度来说，一个典型的软件系统包含一些核心的关注点和系统级的关注点。举个例子来说，一个信用卡处理系统的核心关注点是借贷/存入处理，而系统级的关注点则是日志、事务完整性、授权、安全及性能问题等，许多关注点——即横切关注点（crosscutting concerns）——会在多个模块中出现。如果使用现有的编程方法，横切关注点会横越多个模块，结果是使系统难以设计、理解、实现和演进。AOP能够比上述方法更好地分离系统关注点，从而提供模块化的横切关注点。

例如一个复杂的系统，它由许多关注点组合实现，如业务逻辑、性能，数据存储、日志和调度信息、授权、安全、线程、错误检查等，还有开发过程中的关注点，如易懂、易维护、易追查、易扩展等，图2.1演示了由不同模块实现的一批关注点组成一个系统。

  
图2.1 把模块作为一批关注点来实现

通过对系统需求和实现的识别，我们可以将模块中的这些关注点分为：核心关注点和横切关注点。对于核心关注点而言，通常来说，实现这些关注点的模块是相互独立的，他们分别完成了系统需要的商业逻辑，这些逻辑与具体的业务需求有关。而对于日志、安全、持久化等关注点而言，他们却是商业逻辑模块所共同需要的，这些逻辑分布于核心关注点的各处。在AOP中，诸如这些模块，都称为横切关注点。应用AOP的横切技术，关键就是要实现对关注点的识别。

如果将整个模块比喻为一个圆柱体，那么关注点识别过程可以用三棱镜法则来形容，穿越三棱镜的光束（指需求），照射到圆柱体各处，获得不同颜色的光束，最后识别出不同的关注点。如图2.2所示：

  
图2.2 关注点识别：三棱镜法则

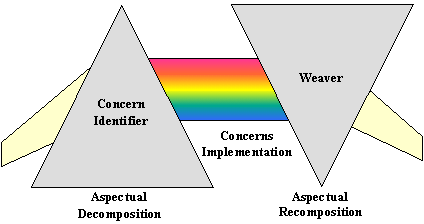
上图识别出来的关注点中，Business Logic属于核心关注点，它会调用到Security，Logging，Persistence等横切关注点。

public class BusinessLogic  
{  
    public void SomeOperation()  
    {  
       //验证安全性；Securtity关注点；  
       //执行前记录日志；Logging关注点；

       DoSomething();

       //保存逻辑运算后的数据；Persistence关注点；  
       //执行结束记录日志；Logging关注点；  
    }  
}

AOP的目的，就是要将诸如Logging之类的横切关注点从BusinessLogic类中分离出来。利用AOP技术，可以对相关的横切关注点封装，形成单独的“aspect”。这就保证了横切关注点的复用。由于BusinessLogic类中不再包含横切关注点的逻辑代码，为达到调用横切关注点的目的，可以利用横切技术，截取BusinessLogic类中相关方法的消息，例如SomeOperation()方法，然后将这些“aspect”织入到该方法中。例如图2.3：

  
图2.3 将横切关注点织入到核心关注点中

通过利用AOP技术，改变了整个系统的设计方式。在分析系统需求之初，利用AOP的思想，分离出核心关注点和横切关注点。在实现了诸如日志、事务管理、权限控制等横切关注点的通用逻辑后，开发人员就可以专注于核心关注点，将精力投入到解决企业的商业逻辑上来。同时，这些封装好了的横切关注点提供的功能，可以最大限度地复用于商业逻辑的各个部分，既不需要开发人员作特殊的编码，也不会因为修改横切关注点的功能而影响具体的业务功能。

为了建立松散耦合的、可扩展的企业系统，AOP应用到的横切技术，通常分为两种类型：动态横切和静态横切。

2.2.2.2 动态横切

动态横切是通过切入点和连接点在一个方面中创建行为的过程，连接点可以在执行时横向地应用于现有对象。动态横切通常用于帮助向对象层次中的各种方法添加日志记录或身份认证。在很多应用场景中，动态横切技术基本上代表了AOP。

动态横切技术的核心主要包括join point（连接点），point cut（切入点），advice（通知）和aspect（方面）。在前面，我已经概要地介绍了这些术语分别代表的含义。接下来，我将以一个具体的实例来进一步阐述它们在AOP动态横切中实现的意义。

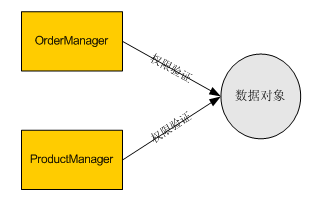
考虑一个电子商务系统，需要对订单进行添加、删除等管理操作。毫无疑问，在实际的应用场景中，这些行为应与权限管理结合，只有获得授权的用户方能够实施这些行为。采用传统的设计方法，其伪代码如下：  
public class OrderManager  
{  
    private ArrayList m\_Orders;  
    public OrderManager()  
    {  
       m\_Orders = new ArrayList();  
    }  
    public void AddOrder(Order order)  
    {  
        if (permissions.Verify(Permission.ADMIN))  
        {

            m\_Orders.Add(order);  
        }  
    }

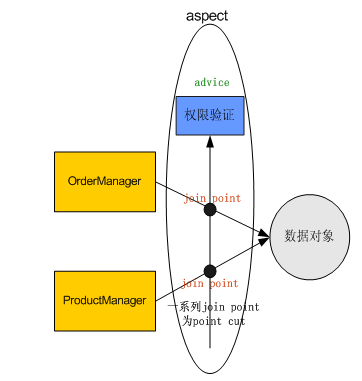
    public void RemoveOrder(Order order)  
    {  
        if (permissions.Verify(Permission.ADMIN))  
        {  
            m\_Orders.Remove(order);  
        }  
    }  
}

同样的，在该电子商务系统中，还需要对商品进行管理，它采用了同样的授权机制：  
public class ProductManager  
{  
    private ArrayList m\_Products;  
    public ProductManager()  
    {  
        m\_Products = new ArrayList();  
    }  
    public void AddProduct(Product product)  
    {  
        if (permissions.Verify(Permission.ADMIN))  
        {  
             m\_Products.Add(product);  
        }  
    }  
    public void RemoveProduct(Product product)  
    {  
        if (permissions.Verify(Permission.ADMIN))  
        {  
             m\_Products.Remove(product);  
        }  
    }  
}

如此以来，在整个电子商务系统中，核心业务包括订单管理和商品管理，它们都需要相同的权限管理，如图2.4所示：

  
图2.4 电子商务系统的权限验证实现

毫无疑问，利用AOP技术，我们可以分离出系统的核心关注点和横切关注点，从横向的角度，截取业务管理行为的内部消息，以达到织入权限管理逻辑的目的。当执行AddOrder()等方法时，系统将验证用户的权限，调用横切关注点逻辑，因此该方法即为AOP的join point。对于电子商务系统而言，每个需要权限验证的方法都是一个单独的join point。由于权限验证将在每个方法执行前执行，所以对于这一系列join point，只需要定义一个point cut。当系统执行到join point处时，将根据定义去查找对应的point cut，然后执行这个横切关注点需要实现的逻辑，即advice。而point cut和advice，就组合成了一个权限管理aspect。

  
图2.5 AOP动态横切的技术实现

由于aspect是一个封装的对象，我们可以定义这样一个aspect：  
private static aspect AuthorizationAspect{……}

然后在这个aspect中定义point cut，在point cut中，定义了需要截取上下文消息的方法，例如：  
private pointcut authorizationExecution():  
execution(public void OrderManager.AddOrder(Order)) ||  
execution(public void OrderManager.DeleteOrder(Order)) ||  
execution(public void ProductManager.AddProduct(Product)) ||  
execution(public void ProductManager.DeleteProduct(Product));

由于权限验证是在订单管理方法执行之前完成，因此在before advice中，定义权限检查：  
before(): authorizationExecution()  
{  
    if !(permissions.Verify(Permission.ADMIN))  
    {  
        throw new UnauthorizedException();  
    }  
}

通过定义了这样一个完整的aspect，当系统调用OrderManager或ProductManager的相关方法时，就触发了point cut，然后调用相应的advice逻辑。如此以来，OrderManager和ProductManager模块就与权限管理模块完全解除了依赖关系，同时也消除了传统设计中不可避免的权限判断的重复代码。这对于建立一个松散耦合、可扩展的系统软件是非常有利的。

2.2.2.3 静态横切

静态横切和动态横切的区别在于它不修改一个给定对象的执行行为。相反，它允许通过引入附加的方法字段和属性来修改对象的结构。此外，静态横切可以把扩展和实现附加到对象的基本结构中。在AOP实现中，通常将静态横切称为introduce或者mixin。

静态横切在AOP技术中，受到的关注相对较少。事实上，这一技术蕴含的潜力是巨大的。使用静态横切，架构师和设计者能用一种真正面向对象的方法有效地建立复杂系统的模型。静态横切允许您不用创建很深的层次结构，以一种本质上更优雅、更逼真于现实结构的方式，插入跨越整个系统的公共行为。尤其是当开发应用系统时，如果需要在不修改原有代码的前提下，引入第三方产品和API库，则静态横切技术将发挥巨大的作用。

举例来说，当前已经实现了一个邮件收发系统，其中类Mail完成了收发邮件的功能。但在产品交付后，发现该系统存在缺陷，在收发邮件时，未曾实现邮件地址的验证功能。现在，第三方产品已经提供了验证功能的接口IValidatable：  
public interface IValidatable  
{  
    bool ValidateAddress();  
}

我们可以利用设计模式中的Adapter模式，来完成对第三方产品API的调用。我们可以定义一个新的类MailAdapter，该类实现了IValidatable接口，同时继承了Mail类：  
public class MailAdapter:Mail,IValidatable  
{  
     public bool ValidateAddress()  
     {  
         if(this.getToAddress() != null)  
         {  
             return true;  
         }  
         else  
         {  
             return false;  
         }  
     }  
}

通过引入MailAdapter类，原来Mail对象完成的操作，将全部被MailAdapter对象取代。然而，此种实现方式虽然能解决引入新接口的问题，但类似下面的代码，却是无法编译通过的：  
Mail mail = new Mail();  
IValidatable validate = ((IValidatable)mail).ValidateAddress();

必须将第一行代码作如下修改：  
Mail mail = new MailAdapter();

利用AOP的静态横切技术，可以将IValidatable接口织入到原有的Mail类中，这是一种非常形象的introduce功能，其实现仍然是在aspect中完成：  
import com.acme.validate.Validatable;

public aspect MailValidateAspect  
{  
    declare parents: Mail implements IValidatable;

    public boolean Mail.validateAddress()  
    {  
         if(this.getToAddress() != null)  
         {  
              return true;  
         }  
         else  
         {  
              return false;  
         }  
    }  
}

静态横切的方法，并没有引入类似MailAdapter的新类，而是通过定义的MailValidateAspect方面，利用横切技术为Mail类introduce了新的方法ValidateAddress()，从而实现了Mail的扩展。因此如下的代码完全可行。  
Mail mail = new Mail();  
IValidatable validate = ((IValidatable)mail).ValidateAddress();

2.3 AOP技术的优势

AOP技术的优势是显而易见的。在面向对象的世界里，人们提出了各种方法和设计原则来保障系统的可复用性与可扩展性，以期建立一个松散耦合、便于扩展的软件系统。例如GOF提出的“设计模式”，为我们提供了设计的典范与准则。设计模式通过最大程度的利用面向对象的特性，诸如利用继承、多态，对责任进行分离、对依赖进行倒置，面向抽象，面向接口，最终设计出灵活、可扩展、可重用的类库、组件，乃至于整个系统的架构。在设计的过程中，通过各种模式体现对象的行为、暴露的接口、对象间关系、以及对象分别在不同层次中表现出来的形态。然而鉴于对象封装的特殊性，“设计模式”的触角始终在接口与抽象中大做文章，而对于对象内部则无能为力。

通过“横切”技术，AOP技术就能深入到对象内部翻云覆雨，截取方法之间传递的消息为我所用。由于将核心关注点与横切关注点完全隔离，使得我们能够独立的对“方面”编程。它允许开发者动态地修改静态的OO模型，构造出一个能够不断增长以满足新增需求的系统，就象现实世界中的对象会在其生命周期中不断改变自身，应用程序也可以在发展中拥有新的功能。

设计软件系统时应用AOP技术，其优势在于：

（一）在定义应用程序对某种服务（例如日志）的所有需求的时候。通过识别关注点，使得该服务能够被更好的定义，更好的被编写代码，并获得更多的功能。这种方式还能够处理在代码涉及到多个功能的时候所出现的问题，例如改变某一个功能可能会影响到其它的功能，在AOP中把这样的麻烦称之为“纠结（tangling）”。

（二）利用AOP技术对离散的方面进行的分析将有助于为开发团队指定一位精于该项工作的专家。负责这项工作的最佳人选将可以有效利用自己的相关技能和经验。

（三）持久性。标准的面向对象的项目开发中，不同的开发人员通常会为某项服务编写相同的代码，例如日志记录。随后他们会在自己的实施中分别对日志进行处理以满足不同单个对象的需求。而通过创建一段单独的代码片段，AOP提供了解决这一问题的持久简单的方案，这一方案强调了未来功能的重用性和易维护性：不需要在整个应用程序中一遍遍重新编写日志代码，AOP使得仅仅编写日志方面（logging aspect）成为可能，并且可以在这之上为整个应用程序提供新的功能。

总而言之，AOP技术的优势使得需要编写的代码量大大缩减，节省了时间，控制了开发成本。同时也使得开发人员可以集中关注于系统的核心商业逻辑。此外，它更利于创建松散耦合、可复用与可扩展的大型软件系统。

<http://wayfarer.cnblogs.com/articles/241024.html>

Java平台AOP技术研究

3.1 Java平台AOP技术概览

3.1.1 AOP技术在Java平台中的应用

AOP在实验室应用和商业应用上，Java平台始终走在前面。从最初也是目前最成熟的AOP工具——AspectJ，到目前已经融和在企业级容器JBoss中的JBoss AOP，均建立在Java平台上。

前面已经描述到，AOP的目的就是将核心关注点和横切关注点分离，实际上这就是一种分散关注（seperation of concerns）的思路。在Java平台下，如果要开发企业级的应用，非J2EE莫属。一个J2EE应用系统只有部署在J2EE容器中才能运行，那么为什么要划分为J2EE容器和J2EE应用系统？ 通过对J2EE容器运行机制的分析，我们发现：实际上J2EE容器分离了一般应用系统的一些通用功能，例如事务机制、安全机制以及对象池或线程池等性能优化机制。这些功能机制是每个应用系统几乎都需要的，因此可以从具体应用系统中分离出来，形成一个通用的框架平台，而且，这些功能机制的设计开发有一定难度，同时运行的稳定性和快速性都非常重要，必须经过长时间调试和运行经验积累而成，因此，形成了专门的J2EE容器服务器产品，如Tomcat JBoss、Websphere、WebLogic等。

从J2EE将应用系统和容器分离的策略，我们能够看到AOP的影子。J2EE应用系统就相当于AOP技术中的核心关注点，它的内容主要包括企业系统的商业逻辑；而J2EE容器则类似于横切关注点，实现的是通用的功能机制。不过业界在选择J2EE容器时，对于EJB这种重量级容器服务器而言，虽然欣赏其高效、稳定及企业级的容器服务，但对于整个容器的高开销、高成本以及过于复杂的解决方案均深怀戒心。因此，随着J2EE的逐步演化，“轻量级容器架构”通过开源社区如激流一般的驱动力，逐渐占据了J2EE技术的强势地位。而所谓“轻量级容器”与EJB提供的重量级架构的区别，就在于借助了AOP技术和IoC（Inversion of Control，反转模式）机制，降低了代码对于专用接口的依赖性，以简短、轻便、专注、可移植的方式实现业务对象。事实上，我们看到的美好前景是，如果所有企业级服务都可以通过AOP机制提供给普通Java对象，那么深盔重铠的应用服务器就不再有存在的价值了。

正是看到了AOP技术在企业级开发中的巨大潜力，而“轻量级容器”也唤起了改革EJB容器的呼声（事实上，最新的 EJB V3.0 标准就使用了轻量级容器模型），越来越多的AOP工具在Java平台下应运而生，从而形成了目前AOP工具百家争鸣的局面。其中，应用最为广泛的主要包括AspectJ、Spring AOP和JBoss AOP等。

3.1.2 Java平台下AOP工具的比较

AOP是一项新技术，而在Java平台下实现该技术的工具也非常多。虽然AOP的技术要素从本质上来讲是一致的，但各种工具的实现方法也各有不同，本节基于AOP的技术要素，对当前应用较广泛的AspectJ、Spring AOP和JBoss AOP进行比较。

3.1.2.1 AOP实现机制的区别

同样是实现AOP，且AOP的技术要素完全相同，但各种AOP工具对于AOP实现的底层机制却是不尽相同的。

AspectJ采用了源代码生成技术来实现AOP。它提供了一套独有的基于Java平台的AOP语法，以及专有的AspectJ编译器。编译器在编译具有AspectJ语法的Java程序时，能够识别诸如aspect，pointcut等特殊关键字，然后利用静态织入的方式，修改需要被截取的方法所属类的源代码，把advice或者introduce的业务逻辑代码注入到正确的位置。利用AspectJ，可以将核心关注点完全独立出来，然后通过AspectJ语法，编写符合核心关注点要求的横切关注点代码，最后通过AspectJ编译器，将这两者在后期结合起来。采用这种静态织入技术，使得运用了AOP技术的系统在运行性能上未受到任何损失，因为它没有利用反射技术或代理技术，而仅仅是程序的静态扩展而已。然而这种源代码生成方式实现的AOP虽然在性能上具备一定的优势，但它同时会给开发带来一定的问题。例如代码的后期修改会给系统带来不可估量的影响。

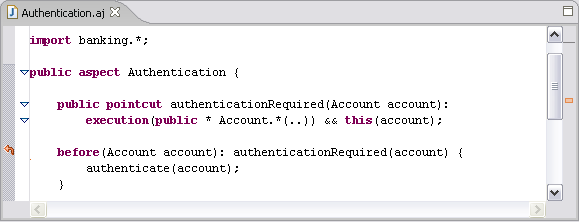
Spring AOP是Spring框架中的一部分，但可以作为一个独立的模块单独存在。Spring AOP实现AOP技术从本质上来讲，是利用了JDK提供的动态代理技术。而从实际的实现方式来看，则是利用了IoC（Inversion of Control，反转模式）机制，同时采用了AOP联盟（AOP Alliance）的通用AOP接口。首先，Spring AOP通过xml配置文件配置了pointcut，并利用Interceptor（拦截机）作为设定的触发条件。Interceptor是由用户自定义的，它相当于是AOP中的advice，但该Interceptor需要实现AOP联盟的通用AOP接口，例如org.aopalliance.intercept.MethodInterceptor。最后定义一个Spring AOP ProxyFactory用于加载执行AOP组件，并利用IoC机制将advice注入到接口以及实现类中。

JBoss 4.0提供了AOP框架。与Spring一样，这个框架可与JBoss应用服务器紧密结合，也可以单独运行在自己的应用中。JBoss AOP同样需要Interceptor拦截器来完成对方法的拦截，它要求所有的Interceptor都必须实现org.jboss.aop.Interceptor接口。在这个接口中最重要的方法就是invoke()。该方法对元数据直接进行操作，并利用反射的原理去拦截方法的消息。Interceptor相当于AOP的advice，至于pointcut，则在xml配置文件中配置。可以看出，Spring AOP和JBoss AOP在实现上属于动态织入的方式，它们与AspectJ在实现上是迥然不同的两种方式。

3.1.2.2 关于“Aspect（方面）”的区别

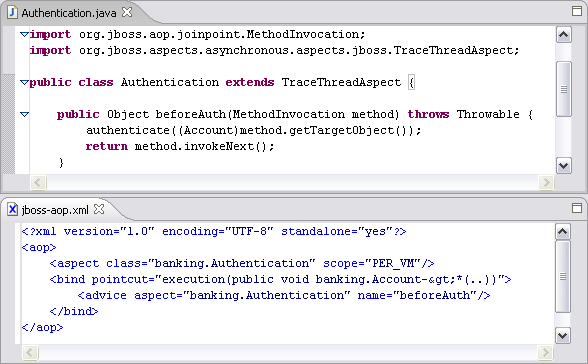
在对aspect的声明上，可以使用类似Java的代码，注释或xml。考虑一个常用的例子，对Account类的授权策略，如果以AOP技术来实现，运用不同的AOP工具，它们在方面声明技术上的差异，是显而易见的。

Aspect 中的方面声明类似于 Java 语言中的类声明，如图3.1 所示。

  
图3.1 AspectJ中的方面声明

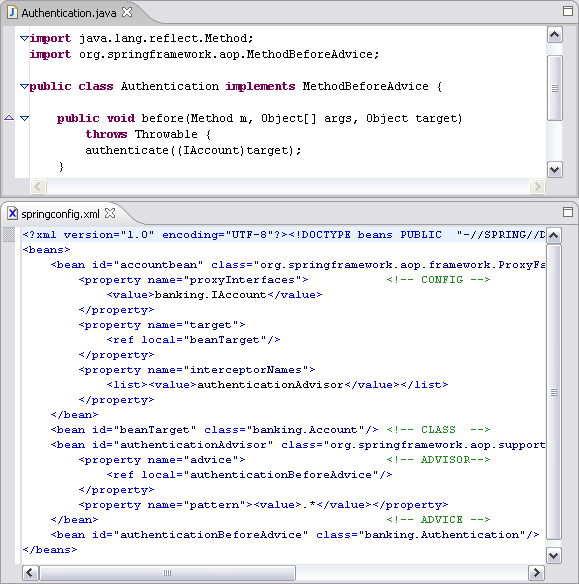
由于 AspectJ 是 Java 语言语法和语义的扩展，所以它提供了自己的一套处理方面的关键字。除了包含字段和方法之外，AspectJ 的方面声明还包含pointcut和advice成员。示例中的pointcut使用了修饰符（modifier）和通配符（wildcard）模式来表达“所有公共方法”。对帐户的访问，由 pointcut 参数提供。advice使用这个参数，而pointcut则用 this(account) 把它绑定。这样做的效果，就是捕获了正在执行的方法所隶属的Account对象。否则，advice的主体与方法的主体相似。advice可以包含认证代码，或者就像在这个示例中一样，可以调用其他方法。

JBoss AOP 基于 XML 的风格来声明方面，如图 3.2 所示。

  
图3.2 JBoss AOP的方面声明

在 XML 风格中，aspect、pointcut和advice的声明都以 XML 形式表示的。advice的实现，用的是普通的 Java 方法，由JBoss AOP框架调用。pointcut和pointcut到advice的绑定都在方面中用XML注释声明。JBoss 没有显式地绑定 Account 参数，而是提供了对当前正在执行的对象的反射访问，因此需要把类型转换到对应的类型。JBoss AOP还可以通过标签的方式对方面进行声明。标签均以“@”字符开始，它的使用有点类似于.Net中的Attribute。

Spring AOP同样是基于 XML 的风格来声明方面，如图3.3所示。

  
图3.3 Spring AOP的方面声明

与JBoss AOP类似，Spring的advice实现是带有特殊参数的Java方法，由 Spring 框架调用。XML描述accountBean，Spring框架通过它访问 Account 对象，包括通知使用的拦截器 advisor 及其匹配模式，还有应用到模式的向前（before） 通知。

由于Spring AOP利用了IoC机制，因此比较JBoss AOP而言，在xml配置文件中提供了更加精细的配置。而构建、运行和配置 Spring AOP 方面的过程则与JBoss AOP基本相同，不过Spring AOP依赖的是Spring框架方便的、最小化的运行时配置，所以不需要独立的启动器。

3.1.2.3 语言机制的区别

    由于实现机制和语法风格的不同，三种AOP工具在语言机制上也有很大的不同，以下从四个方面来描述AspectJ、JBossAOP和Spring AOP之间的区别。

（1）pointcut匹配和复合：AspectJ和 JBoss AOP 提供了类似的类型模式支持。它们都允许签名方面的匹配，对于 Java 5 应用程序来说，这些匹配包括注释和泛型。AspectJ提供了一种简洁的引用多个类型的技术（例如 Account+ 表示帐户的所有子类型）。所有的工具都支持通配符匹配。Spring AOP 还提供了对正则表达式的支持。虽然这看起来可能是一个强大的优势，但还是要指出其他技术已经选择了放弃正则表达式，好让pointcut读起来不是太难，同时不会存在潜在的损害。pointcut复合操作符基本上都是相同的。Spring AOP 不提供“非”操作，这个操作通常与没有在 Spring AOP 连接点模型的容器（containment）连接点结合使用。

（2）advice形式：AspectJ 支持比其他技术更多的advice形式，而 JBoss AOP 只支持一种advice形式。每种通知形式都可以表达成 around advice，所以 JBoss 的技术是无限的，而且它确实提供了额外的简单性。不好的一面是它损失了简洁性。另外，强迫advice去遵守普通的 Java 规则（就像注释和 XML 风格做的那样），在一些情况下容易出问题，因为这些规则是为方法设计的。AspectJ 拥有把被通知方法的异常“软化”的能力，这很有用，但是不符合方法异常检测的标准语义。

（3）join point上下文：在 AspectJ中，通过指定和绑定pointcut参数访问动态连接点的状态，类似于在 Java 语言中声明方法参数的技术（请参阅图3.1）。这为连接点上下文提供了静态类型化的好处。JBoss AOP 和 Spring AOP 反射性地访问连接点的状态，这消除了在切入点表达式中参数绑定的复杂性，代价是参数静态类型化。Java 程序员习惯了方法参数静态类型化带来的好处，同时还可以从pointcut参数的静态类型化得到同样的好处。所以，在 JBoss AOP 最近的发行版本中，有提供静态类型化的“args”的计划。

（4）扩展性：aspect的扩展性支持库方面的部署，这样可以在日后为特定程序将这些库方面具体化。例如，一个方面库可以提供应用程序监视需要的全部逻辑和基础设施。但是，要采用某个特定项目的库，那么库使用的pointcut必须扩展成应用程序特定的join point。AspectJ 用抽象方面支持扩展性，抽象方面包含抽象的pointcut和具体的advice。扩展抽象方面的子方面必须具体化pointcut。JBoss AOP 使用了完全不同的技术，没有使用抽象切入点机制。扩展是通过生成aspect的子类、并在 XML 中或通过注释定义新的advice绑定而实现的。pointcut到advice的显式绑定为JBoss AOP提供了显著优势，从而可以很容易地把方面扩展到新系统，无需要生成子类。

3.2 Java平台下AOP主流工具研究

3.2.1 AsepctJ研究

AspectJ作为Java编程语言扩展的AOP工具，使得我们运用AOP技术能够像普通的Java编程那样，特殊之处，仅在于我们需要使用AspectJ提供的特殊语法。接下来，我将通过一些实例，介绍如何运用AspectJ实现AOP技术。

3.2.1.1 AspectJ语言特性

设定我们的开发项目中需要应用到日志记录，根据前面介绍的AOP知识，我们已经能够从这个需求中识别出横切关注点——日志记录。因此，我们需要定义关于“日志记录”的aspect：

public aspect AutoLog  
{   
    pointcut publicMethods() : execution(public \* org.apache.cactus..\*(..));  
    pointcut logObjectCalls() : execution(\* Logger.\*(..));  
    pointcut loggableCalls() : publicMethods() && ! logObjectCalls();

    before() : loggableCalls()  
    {  
      Logger.entry(thisJoinPoint.getSignature().toString());  
    }  
    after() : loggableCalls()  
    {  
      Logger.exit(thisJoinPoint.getSignature().toString());  
    }  
}

如果仅仅熟悉Java编程，会发现有很多关键字是Java语言中不曾包含的，它们均是AspectJ提供的。

分析上述的代码，首先是aspect的声明，它类似于Java中的类声明，定义了一个aspect：AutoLog。在这个方面中分别包含了pointcut和advice。

pointcut共有三个：publicMethod、logObjectCalls和loggableCalls。publicMethod将选择org.apache.cactus包中的所有公共（public）方法的执行。所谓“选择”，就意味着它的join point为其选择的方法。当这些方法被调用时，就会执行pointcut的advice代码。而在pointcut中，execution 是一个原始的 Pointcut（就象 int 是一种原始的 Java 类型）。它选择与括号中定义的方法说明匹配的任何方法的执行。方法说明允许包含通配符。logObjectCalls的pointcut则选择Logger 类中的所有方法的执行。第三个pointcut比较特殊，它使用&& !合并了前两个 Pointcut，这意味着它选者了除Logger类中的公共方法以外， org.apache.cactus 中所有的公共方法。

advice在aspect中，被用来完成实际的日志纪录。advice有三种，分别为before、after和around。如上述代码中定义的advice：  
before() : loggableCalls()  
{  
    Logger.entry(thisJoinPoint.getSignature().toString());  
}

该advice的定义表示的含义是，如果org.apache.cactus中所有的公共方法（Logger类的公共方法除外）被执行，则在这些方法执行之前，需要先执行该advice定义的逻辑。

3.2.1.2 AspectJ的高级语言特性

在本文第二部分介绍AOP技术时，提到了横切技术的分类。其中，静态横切技术能够扩展一个对象的结构。使用引入（Introduction），Aspect 可以向类中添加新的方法和变量、声明一个类实现一个接口或将检查异常转换为未检查异常（unchecked exception）。

3.2.1.2.1 向现有类添加变量和方法

假设您有一个表示持久存储的数据缓存的对象。为了测量数据的“更新程度”，您可能决定向该对象添加时间戳记字段，以便容易地检测对象是否与后备存储器同步。由于对象表示业务数据，根据AOP的知识，我们应该将这种机制性细节从对象中隔离。使用 AspectJ，可以用如下代码中所显示的语法来向现有的类添加时间戳记：

public aspect Timestamp  
{  
    private long ValueObject.timestamp;  
    public long ValueObject.getTimestamp()  
    {  
       return timestamp;  
    }  
    public void ValueObject.timestamp()  
    {       
       this.timestamp = System.currentTimeMillis();  
    }  
}

通过introduction，我们就非常方便的为ValueObject类型添加了timestamp的变量和相关方法。除了必须限定在哪个类上声明引入的方法和成员变量以外，声明引入的方法和成员变量几乎与声明常规类成员相同。

3.2.1.2.2实现多继承功能

利用introduction，AspectJ允许向接口和类添加成员，也突破了Java语言只能单继承的限制，允许程序按C++方式那样实现多继承。如果您希望上述的aspect Timestamp能够泛化 （generalize），以便能够对各种对象重用时间戳记代码，可以定义一个称为 TimestampedObject 的接口，并使用引入（Introduction）来将相同成员和变量添加到接口而不是添加到具体类中，如下所示:  
public interface TimestampedObject  
{  
    long getTimestamp();  
    void timestamp();  
}  
public aspect Timestamp  
{  
    private long TimestampedObject.timestamp;  
    public long TimestampedObject.getTimestamp()  
    {  
        return timestamp;  
    }  
    public void TimestampedObject.timestamp()  
    {  
        this.timestamp = System.currentTimeMillis();  
    }  
}

Timestamp方面由于在TimestampedObject接口中引入（introduction）了方法的实现，使得TimestampedObject接口改变其本质，成为了一个特殊的类类型。特殊之处就在于一个已经继承了一个类的类类型，通过AspectJ的语法，仍然可以再次继承TimestampedObject，这就间接地实现了类的多继承。而这个特殊的AspectJ语法就是declare parents语法。declare parents和其它AspectJ 类型表达一样，可以同时应用于多个类型：  
declare parents: ValueObject || BigValueObject implements TimestampedObject;

3.2.1.3 编译器及工具支持

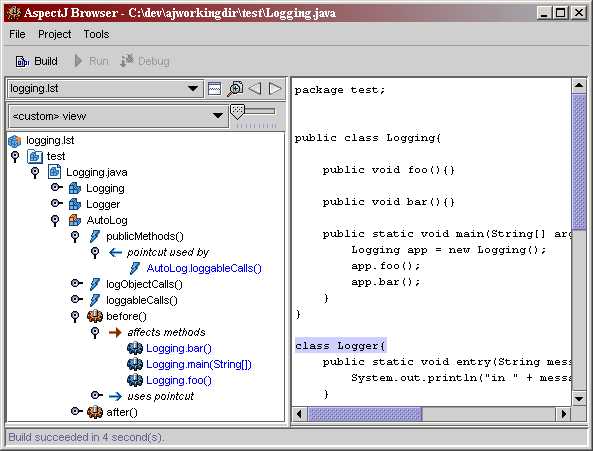
    要让aspect能够正常工作，必须将aspect加入到它们要修改的代码中去。这项工作由AspectJ提供的ajc编译器完成。ajc 编译器用来编译类和 Aspect 代码。ajc 既可以作为编译器也可以作为预编译器操作，生成有效的 .class 或 .java 文件，可以在任何标准 Java 环境（添加一个小的运行时 JAR）中编译和运行这些文件。

要使用 AspectJ 进行编译，将需要显式地指定希望在给定编译中包含的源文件（Aspect 和类），ajc不象javac那样简单地为相关导入模块搜索类路径。之所以这样做，是因为标准 Java 应用程序中的每个类都是相对分离的组件。为了正确操作，一个类只要求其直接引用的类的存在。Aspect 表示跨越多个类的行为的聚集。因此，需要将 AOP 程序作为一个单元来编译，而不能每次编译一个类。

AspectJ 当前版本的一个重要限制是其编译器只能将aspect加入到它拥有源代码的代码中。也就是说，不能使用ajc将Advice添加到预编译类中。AspectJ 团队认为这个限制只是暂时的，AspectJ 网站承诺未来的版本（正式版 2.0）将允许字节码的修改。

AspectJ发行版包含了几种开发工具。这预示着 AspectJ 将有美好的前景，因为它表明了作者对这一部分的一个重要承诺，使 AspectJ 对于开发人员将是友好的。对于面向 Aspect 的系统工具支持尤其重要，因为程序模块可能受到它们所未知的模块所影响。

随 AspectJ 一起发布的一个最重要的工具是图形结构浏览器，它展示了 Aspect 如何与其它系统组件交互。这个结构浏览器既可以作为流行的 IDE 的插件，也可以作为独立的工具。图3.4显示了先前讨论的日志记录示例的视图。

  
图3.4 AspectJ提供的“结构浏览器”工具

除了结构浏览器和核心编译器之外，您还可以从 AspectJ 网站下载一个 Aspect 支持的调试器、一个javadoc工具、一个Ant任务以及一个Emacs 插件。

3.2.2 JBoss AOP研究

JBoss AOP关于AOP的实现与AspectJ是两种完全不同的风格。由于Java利用元数据来存储有关类型、方法、字段的相关信息，因此，可以通过Java提供的反射功能获得模块相关的元数据，对方法进行拦截，并将被拦截的方法与aspect逻辑进行关联。

3.2.2.1 拦截器（Interceptor）

在JBoss AOP中，是用拦截器来实现advice的。可以自定义拦截器，拦截方法调用、构造函数调用以及对字段的访问，但JBoss要求这些自定义的拦截器，必须实现org.jboss.aop.Interceptor接口：

public interface Interceptor  
{  
    public String getName();  
    public InvocationResponse invoke(Invocation invocation) throws Throwable;  
}

在JBoss　AOP中，被拦截的字段、构造器和方法均被转化为通用的invoke方法调用。方法的参数接收一个Invocation对象，而方法的返回值、字段的存取以及构造函数则被填入一个InvocationResponse对象。Invocation对象同时还驱动拦截链。下面我们自定义一个拦截器，它能够拦截构造函数和方法的调用，并将跟踪信息打印到控制台上：  
import org.jboss.aop.\*;  
import java.lang.reflect.\*;

public class TracingInterceptor implements Interceptor  
{  
    public String getName()  
    {  
        return TracingInterceptor;  
    }  
    public InvocationResponse invoke(Invocation invocation) throws Throwable  
    {  
        String message = null;  
        if (invocation.getType() == InvocationType.METHOD)  
        {  
            Method method = MethodInvocation.getMethod(invocation);  
            message = method: + method.getName();  
        }  
        else  
        {  
            if (invocation.getType() == InvocationType.CONSTRUCTOR)  
            {  
                Constructor c = ConstructorInvocation.getConstructor(invocation);  
                message = constructor: + c.toString();  
            }  
            else  
            {  
                // 不对字段作处理，太繁琐；  
                return invocation.invokeNext();  
            }  
            System.out.println(Entering + message);  
        }  
        // 继续。调用真正的方法或者构造函数  
        InvocationResponse rsp = invocation.invokeNext();  
        System.out.println(Leaving + message);  
        return rsp;  
    }  
}

在自定义的TracingInterceptor类中，invoke()方法对invocation的类型作判断，以根据方法、构造函数和字段类型，分别作出不同的操作。而其中，invocation.invokeNext()则表示通过一个拦截链获得下一个invocation。

定义的拦截必须在xml文件配置，使其绑定到具体的类。这个定义即为AOP中的切入点pointcut。例如具体的类为BusinessObject，则该pointcut在xml中的定义如下：  
<?xml version="1.0" encoding="UTF-8">  
<aop>  
    <interceptor-pointcut class="BusinessObject">  
        <interceptors>  
            <interceptor class="TracingInterceptor" />  
        </interceptors>  
    </interceptor-pointcut>  
</aop>

上面的pointcut绑定TracingInterceptor到一个叫做BusinessObject的类。如果要将该Interceptor绑定到多个类，还可以利用正则表达式。例如，如果你想绑定由JVM载入的类，类表达式将变为 .\*。如果你仅仅想跟踪一个特定的包，那么表达式将是bruce.zhang.mypackge.\*。

当JBoss AOP独立运行时，任何符合 META-INF/jboss-aop.xml模式的XML文件将被JBoss AOP 运行期程序载入。如果相关的路径被包含在任何JAR或你的CLASSPATH目录中，该XML文件将在启动时，由JBoss AOP 运行期程序载入。

JBoss AOP还提供了过滤功能，可以通过在xml文件中配置过滤的标志，使一些特定的方法（包括字段的访问）被过滤，从而不再执行Interceptor的相关逻辑。例如，我们要过滤BusinessObject类的所有get()和set()方法，以及main()方法，则可以修改上述的xml文件：  
<?xml version="1.0" encoding="UTF-8">  
<aop>  
    <class-metadata group="tracing" class=" BusinessObject ">  
        <method name="(get.\*)|(set.\*)">  
            <filter>true</filter>  
        </method>  
        <method name="main">  
            <filter>true</filter>  
        </method>  
    </class-metadata>  
</aop>

相应的，Interceptor代码也应作相关的修改，使其能够识别配置文件中的filter属性：  
public class TracingInterceptor implements Interceptor  
{  
    ……//getName()方法略；  
    public InvocationResponse invoke(Invocation invocation) throws Throwable  
    {  
        String filter=(String)invocation.getMetaData(tracing, filter);  
        if (filter != null && filter.equals(true))  
            return invocation.invokeNext();  
        ……//后面的代码略；  
    }  
}

3.2.2.2 引入（Introduction）

JBoss AOP同样提供introduction功能，通过它，就可以为现有的类引入第三方接口或类的API了。例如，我们可以为具体的类如BusinessObject提供Tracing的开关，使得BusinessObject对象能够根据具体的情况打开或关闭aspect的Tracing功能。为实现该功能，可以定义一个Tracing接口：  
public interface Tracing  
{  
    void enableTracing();  
    void disableTracing();  
}

接下来需要定义一个混合类，它实现了接口Tracing。当BusinessObject类被实例化时，该混合类的实例就会被绑定到BusinessObject上。实现方法如下：  
import org.jboss.aop.Advised;

public class TracingMixin implements Tracing  
{  
    Advised advised;

    Public TracingMixin(Object obj)  
    {  
        this.advised = (Advised)obj;  
    }  
    public void enableTracing()  
    {  
        advised.\_getInstanceAdvisor().getMetaData().addMetaData("tracing", "filter", true);  
    }  
    public void disableTracing()  
    {  
        advised.\_getInstanceAdvisor().getMetaData().addMetaData("tracing", "filter", false);  
    }  
}

enableTracing()方法将filter属性绑定到对象实例。disableTracing()方法作同样的事，但是将filter属性设置为false。

定义了Tracing接口和实现了该接口的混合类后，就可以在xml文件中定义一个pointcut，强制BusinessObject类实现Tracing接口：  
<?xml version="1.0" encoding="UTF-8">  
<aop>  
    <introduction-pointcut class="BusinessObject">  
        <mixin>  
            <interfaces>Tracing</interfaces>  
            <class>TracingMixin</class>  
            <construction>new TracingMixin(this)</construction>  
        </mixin>  
    </introduction-pointcut>  
</aop>

注意xml文件中的标签，它代表的含义是当BusinessObject对象被实例化时，将执行该标签内的代码。以本例而言，当创建BusinessObject对象时，一个TracingMixin类的实例将被创建。任何单行的Java代码都可以放到标签中。

通过“引入（introduction）”功能，在处理BusinessObject对象时，就可以视其为Tracing接口类型而进行操作了，如下的示例：

public class BusinessObject  
{  
    public BusinessObject () {}  
    public void helloWorld() { System.out.println(Hello World!); }

    public static void main(String[] args)  
    {  
        BusinessObject bo = new BusinessObject ();  
        Tracing trace = (Tracing)this;  
        bo.helloWorld();  
         
        System.out.println("Turn off tracing.");  
        trace.disableTracing();  
        bo.helloWorld();  
         
        System.out.println("Turn on tracing.");  
        trace.enableTracing();  
        bo.helloWorld();  
    }  
}

注意如下代码：  
Tracing trace = (Tracing)this;

此时this代表的即为BusinessObject，从Java代码的角度来看，由于BusinessObject并没有实现Tracing接口，因此这行代码所示的显式转换为Tracing类型是不成功的。但通过“引入”功能，使得BusinessObject通过混合类，实现了Tracing接口，从而使得如上的代码能够顺利执行。隐含的意义就是，我们没有修改BusinessObject的定义，而是通过AOP技术，为BusinessObject扩展实现了第三方提供的接口Tracing。

3.2.3 Spring AOP研究

Spring AOP使用纯Java实现，不需要特别的编译过程，也不需要控制类装载层次。与JBoss AOP相同，它仍然利用了拦截器完成对方法的拦截。然而，Spring AOP实现AOP的主要技术却主要来自于AOP联盟，如拦截器应实现org.aopalliance.intercept.MethodInterceptor 接口，而所有advice必须实现org.aopalliance.aop.Advice标签接口。此外，Spring实现AOP的目标也不同于其他大部分AOP框架，它的目标不是提供及其完善的AOP实现，而是提供一个和Spring IoC紧密整合的AOP实现，帮助解决企业应用 中的常见问题。因此，Spring AOP的功能通常是和Spring IoC容器联合使用的。AOP Advice是用普通的bean定义语法来定义的，Advice和pointcut本身由Spring IoC 管理。这是一个重要的其他AOP实现的区别。

3.2.3.1 切入点（pointcut）

Spring的切入点模型能够使pointcut独立于advice类型被重用。同样的pointcut有可能接受不同的advice。将Pointcut接口分成两个部分有利于重用类和方法的匹配部分，并且组合细粒度的操作（如和另一个方法匹配器执行一个“并”的操作）。

在Spring的切入点中，org.springframework.aop.Pointcut接口是重要的接口，它用来指定通知到特定的类和方法目标。完整的接口定义如下:  
public interface Pointcut  
{  
    ClassFilter getClassFilter();  
    MethodMatcher getMethodMatcher();  
}

ClassFilte类型也是一个接口，该接口被用来将切入点限制到一个给定的目标类的集合。 如果matches()永远返回true，所有的目标类都将被匹配。  
public interface ClassFilter  
{  
    boolean matches(Class clazz);  
}

MethodMatcher接口通常更加重要。完整的接口定义如下：  
public interface MethodMatcher  
{  
    boolean matches(Method m, Class targetClass);  
    boolean matches(Method m, Class targetClass, Object[] args);  
    boolean isRuntime();  
}

matches(Method, Class) 方法被用来测试这个切入点是否匹配目标类的给定方法。这个测试可以在AOP代理创建的时候执行，避免在所有方法调用时都需要进行 测试。如果2个参数的matches()方法对某个方法返回true，并且MethodMatcher的isRuntime()也返回true，那么3个参数的matches()方法将在每次方法调用的时候被调用。这使切入点能够在目标advice被执行之前立即查看传递给方法调用的参数。由于大部分MethodMatcher都是静态的，意味着isRuntime()方法会返回false。此种情况下，3个参数的matches()方法永远不会被调用。

Spring AOP提供了几个实用的切入点实现，其中较为常用的是正则表达式切入点：org.springframework.aop.support.RegexpMethodPointcut，它使用Perl 5的正则表达式的语法。使用这个类你可以定义一个模式的列表。如果任何一个匹配，那个切入点将被计算成 true。用法如下：  
<bean id="settersAndAbsquatulatePointcut"  
    class="org.springframework.aop.support.RegexpMethodPointcut">  
    <property name="patterns">  
        <list>  
            <value>.\*get.\*</value>  
            <value>.\*absquatulate</value>  
        </list>  
    </property>  
</bean>

不过，更多情况下是直接使用RegexpMethodPointcut一个实用子类： RegexpMethodPointcutAdvisor。它允许我们同时引用一个advice（在Spring AOP中，advice可以是拦截器，也可以是before advice，throws advice等)。这就简化了bean的装配，因为一个bean可以同时当作pointcut和advice，如下所示：  
<bean id="myPointcutAdvisor" class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">  
    <property name="advice">  
        <ref local="MyInterceptor" />  
    </property>  
    <property name="patterns">  
        <list>  
            <value>.\*save.\*</value>  
            <value>.\*do.\*</value>  
        </list>  
    </property>  
</bean>

注意配置文件中的myPointcutAdvisor，在Spring AOP中，一个advisor就是一个aspect完整的模块化表示。通过advisor，可以将pointcut和advice（在此处即为MyInterceptor）绑定起来。

3.2.3.2 通知（advice）

Spring AOP的advice可以跨越多个被advice对象共享，或者每个被advice对象有自己的advice。要实现advice，最简单的做法就是定义一个拦截器（Interceptor）。它采用了AOP联盟（AOP Alliance）的通用AOP接口（接口定义为aopalliance.jar）。要实现advice，需要实现aopalliance.jar中定义的MethodInterceptor接口。

例如，我们定义了一个业务对象接口BusinessObject及其实现类BusinessObjectImpl，该业务对象能够存储数据，其定义如下：  
public interface BusinessObject  
{  
    public void save();  
}  
public class BusinessObjectImpl implements BusinessObject  
{  
    public void save()  
    {  
         System.out.println("saving domain object......");  
    }  
}

现在需要为业务对象BusinessObject的Save()方法，提供Lock机制。根据Spring AOP的实现方式，我们可以定义一个LockInterceptor来实现MethodInterceptor接口：  
import org.aopalliance.intercept.MethodInterceptor;  
import org.aopalliance.intercept.MethodInvocation;

public class LockInterceptor implements MethodInterceptor  
{  
    public Object invoke(MethodInvocation invocation) throws Throwable  
    {  
        // TODO Auto-generated method stub  
        lock();  
        Object ret= invocation.proceed();  
        unlock();  
        return ret;  
    }  
    private void lock()  
    {  
        System.out.println("lock domain object...");  
    }  
    private void unlock()  
    {  
        System.out.println("unlock domain object...");  
    }  
}

为将interceptor与具体的advice绑定起来，需要在配置文件中配置bean：  
&lt;bean id="MyInterceptor" class="test.aop.spring.LockInterceptor"/&gt;

3.2.3.3 AOP代理与IoC容器

由于Spring中提供了IoC容器（例如BeanFactory），因此我们可以通过Ioc机制，利用ProxyFactoryBean来创建AOP代理。ProxyFactoryBean和其他Spring的 FactoryBean实现一样，引入一个间接的层次。如果你定义一个名字为foo的ProxyFactoryBean，引用foo的对象所看到的不是ProxyFactoryBean实例本身，而是由实现ProxyFactoryBean的类的 getObject()方法所创建的对象。这个方法将创建一个包装了目标对象 的AOP代理。

AOP代理利用的是Java的动态代理技术，通过它就可以加载并执行AOP组件。同时，还需要通过IoC的方式将advice注入到接口以及其实现类。以前面的业务对象BusinessObject为例，在xml配置文件中的配置如下：  
<bean id="myAOPProxy" class="org.springframework.aop.framework.ProxyFactoryBean">  
    <property name="proxyInterfaces">  
       <value>test.aop.spring.BusinessObject</value>  
    </property>  
    <property name="target">  
       <ref local="impl" />  
    </property>  
    <property name="interceptorNames">  
       <value>myPointcutAdvisor</value>  
    </property>  
</bean>  
<bean id="impl" class="test.aop.spring.BusinessObjectImpl"/>

通过上述对pointcut、advice、advisor和AOP代理的配置，我们就可以轻易地在Spring中实现AOP，例如：  
import org.springframework.context.ApplicationContext;  
import org.springframework.context.support.FileSystemXmlApplicationContext;

public class App  
{  
    private BusinessObject bo = null;  
    public static void main(String[] args)  
    {  
        ApplicationContext ctx=new FileSystemXmlApplicationContext("Bean.xml");  
        bo= (BusinessObject) ctx.getBean("myAOPProxy");  
        bo.save();  
    }  
}

首先，通过AOP代理获得BusinessObject对象。当调用BusinessObject对象的save()方法时，拦截器LockInterceptor根据RegexpMethodPointcutAdvisor配置的pointcut和advice之间的关系，判定该方法的调用为join point，从而拦截该方法调用，并注入advice的执行逻辑，即lock()和unlock()，最终实现了AOP。

3.2.3.4 引入（introduction）

在Spring AOP中，将introduction当作advice来处理。与一般的advice一样，introduction advice相当于一种特殊类型的拦截通知，需要实现IntroductionAdvisor和IntroductionInterceptor接口，而IntroductionInterceptor接口继承自MethodInterceptor：  
public interface IntroductionInterceptor extends MethodInterceptor  
{  
    boolean implementsInterface(Class intf);  
}

Introduction通知不能被用于任何pointcut，因为它只能作用于类层次上，而不是方法。我们可以只用InterceptionIntroductionAdvisor来实现导入通知，它有下面的方法：  
public interface InterceptionIntroductionAdvisor extends InterceptionAdvisor  
{  
    ClassFilter getClassFilter();  
    IntroductionInterceptor getIntroductionInterceptor();  
    Class[] getInterfaces();  
}

接下来，我以JBoss AOP一节中的例子来说明introduction在Spring AOP中的应用。我们的目标仍然是为一个已有的业务对象引入第三方接口Tracing：  
public interface Tracing  
{  
    void enableTracing();  
    void disableTracing();  
    boolean enabled();  
}

首先，我们需要一个做大量转化的IntroductionInterceptor。在这里，我们继承 org.springframework.aop.support.DelegatingIntroductionInterceptor 实现类。当然我们可以直接实现IntroductionInterceptor接口，但是大多数情况下 DelegatingIntroductionInterceptor是最合适的。

DelegatingIntroductionInterceptor的设计是将introduction委托到真正实现introduction接口的接口，隐藏完成这些工作的拦截器。委托可以使用构造方法参数设置到任何对象中；默认的委托就是自己（当无参数的构造方法被使用时）。这样在下面的例子里，委托是DelegatingIntroductionInterceptor的子类 TracingMixin。给定一个委托（默认是自身）的 DelegatingIntroductionInterceptor实例寻找被这个委托（而不是IntroductionInterceptor）实现的所有接口，并支持它们中任何一个导入。子类如TracingMixi也可能调用suppressInterflace(Class intf) 方法来隐藏不应暴露的接口。然而，不管IntroductionInterceptor 准备支持多少接口，IntroductionAdvisor将控制哪个接口将被实际暴露。一个导入的接口将隐藏目标的同一个接口的所有实现。

这样，TracingMixin继承DelegatingIntroductionInterceptor并自己实现接口Tracing。父类自动选择支持introduction的Tracing，所以我们不需要指定它。用这种方法我们可以导入任意数量的接口。  
public class TracingMixin extends DelegatingIntroductionInterceptor implements Tracing  
{  
    private boolean enabled;  
    public void enableTracing ()  
    {  
        this.enabled = true;  
    }

    public void disableTracing ()  
    {  
        this. enabled = false;  
    }

    public boolean enabled()  
    {  
        return this.enabled;  
    }  
    public Object invoke(MethodInvocation invocation) throws Throwable  
    {        
        return super.invoke(invocation);  
    }  
}

通常不要需要改写invoke()方法：实现DelegatingIntroductionInterceptor就足够了，如果是引入的方法，DelegatingIntroductionInterceptor实现会调用委托方法， 否则继续沿着连接点处理。

所需的introduction advisor是很简单的。只需保存一个独立的TracingMixin实例，并指定导入的接口，在这里就是Tracing。此时，TracingMixin没有相关配置，所以我们简单地使用new来创建它。

public class TracingMixinAdvisor extends DefaultIntroductionAdvisor  
{  
    public TracingMixinAdvisor() {  
        super(new TracingMixin(),Tracing.class);  
    }  
}

我们可以非常简单地使用这个advisor。它不需要任何配置。（但是，有一点是必要的：就是不可能在没有IntroductionAdvisor 的情况下使用IntroductionInterceptor。） 和引入一样，通常 advisor必须是针对每个实例的，并且是有状态的。我们会有不同的TracingMixinAdvisor。每个被通知对象，会有不同的TracingMixin。advisor组成了被通知对象的状态的一部分。

在Spring中，Spring AOP的核心API已经基本稳定了。和Spring的其它部分一样， AOP框架是模块化的，在保留基础设计的同时提供扩展。在Spring 1.1到1.2阶段有很多地方可能会有所提高，但是这些地方也保留了向后兼容性。它们是：

（一）性能的提高：AOP代理的创建由工厂通过策略接口处理。因此能够支持额外的AOP 代理类型而不影响用户代码或核心实现。  
（二）更具表达力的pointcut：Spring目前提供了一个具有表达力的切入点接口，同时添加了更多的切入点实现。Spring正在考虑提供一个简单但具有强大表达式语言的实现。

<http://wayfarer.cnblogs.com/articles/241031.html>

# **[.Net平台AOP技术研究](http://www.cnblogs.com/wayfarer/articles/256909.html)**

4.1.Net平台AOP技术概览

.Net平台与Java平台相比，由于它至今在服务端仍不具备与unix系统的兼容性，也不具备类似于Java平台下J2EE这样的企业级容器，使得.Net平台在大型的企业级应用上，常常为人所诟病。就目前而言，.Net平台并没有提供AOP技术的直接实现，而微软在未来对于.Net的发展战略目标，我们仍未可知。但我相信微软对于目前炙手可热的AOP技术应该不会视而不见。也许在未来的.Net平台下，会出现类似于Spring那样的轻量级IoC容器，加上O/R Mapping的进一步实现与完善，随着Windows Server操作系统的逐步推新，.Net平台对于企业级系统开发的支持会越来越多。

AOP技术在.Net平台中的应用，相较于Java平台而言，还远不够成熟，功能也相对较弱，目前能够投入商用的AOP工具几乎没有。借鉴Java开源社区的成功，.Net平台下AOP工具的开发也都依托于开源社区的力量。众多开源爱好者，仍然在坚持不懈对AOP技术进行研究和实践，试图找到AOP技术与.Net之间的完美结合点，从而开发出真正能够商用的功能强大的AOP工具。就目前而言，大部分在.Net平台下的AOP工具，大部分均脱胎于Java平台下的AOP工具，例如Spring.Net之于Spring，Eos之于AspectJ。由于Java平台和.Net平台在语言机制上的相似性，使得它们在实现AOP的技术机制上，大体相似，无非是利用静态织入或动态织入的方式，完成对aspect的实现。

目前在.Net平台下的AOP大部分仍然处于最初的开发阶段，各自发布的版本基本都是beta版。其中较有代表性的AOP工具包括Aspect#，Spring.Net，Eos等。

Aspect#是基于Castle动态代理技术实现的。Castle动态代理技术利用了.Net的Emit技术，生成一个新的类去实现特定的接口，或者扩展一个已有的类，并将其委托指向IInterceptor接口的实现类。通过Castle动态代理技术，就可以拦截方法的调用，并将Aspect的业务逻辑织入到方法中。利用Castle动态代理技术，最大的缺陷是它只对虚方法有效，这限制了Aspect#的一部分应用。

Spring.Net从根本意义上来说，是对Spring工具从Java平台向.Net平台的完全移植。它在AOP的实现上与Spring几乎完全相似，仍然利用了AOP联盟提供的拦截器、Advice等实现AOP。Spring.Net的配置文件也与Spring相同。

Eos采用的是静态织入的技术。它提供了独有的编译器，同时还扩展了C#语法，以类似于AspectJ的结构，规定了一套完整的AOP语法，诸如aspect，advice，before，after，pointcut等。Eos充分的利用了.Net中元数据的特点，以IL级的代码对方面进行织入，这也使得它的性能与其他AOP工具比较有较大的提高。

4.2 .Net平台下实现AOP的技术基础

如前所述，在.Net平台下实现AOP，采用的方式主要是静态织入和动态织入的方式。在本文中，我将充分利用.Net的技术特性，包括元数据、Attribute、.Net Remoting的代理技术，将其综合运用，最终以动态织入的方式实现AOP公共类库。本节将介绍实现AOP所必需的.Net知识。

4.2.1元数据（metadata）  
4.2.1.1元数据概述

元数据是一种二进制信息，用以对存储在公共语言运行库（CLR）中可移植可执行文件 (PE) 或存储在内存中的程序进行描述。在.Net中，如果将代码编译为 PE 文件时，便会将元数据插入到该文件的一部分中，而该代码被编译成的Microsoft 中间语言 (MSIL)，则被插入到该文件的另一部分中。在模块或程序集中定义和引用的每个类型和成员都将在元数据中进行说明。执行代码时，运行库将元数据加载到内存中，并引用它来发现有关代码的类、成员、继承等信息。

元数据以非特定语言的方式描述在代码中定义的每一类型和成员。它存储的信息包括程序集的信息，如程序集的版本、名称、区域性和公钥，以及该程序集所依赖的其他程序集；此外，它还包括类型的说明，包括类型的基类和实现的接口，类型成员（方法、字段、属性、事件、嵌套的类型）。

在.Net Framework中，元数据是关键，该模型不再需要接口定义语言 (IDL) 文件、头文件或任何外部组件引用方法。元数据允许 .NET 语言自动以非特定语言的方式对其自身进行描述，此外，通过使用Attribute，可以对元数据进行扩展。元数据具有以下主要优点：

1. 自描述文件

公共语言运行库（CLR）模块和程序集是自描述的。模块的元数据包含与另一个模块进行交互所需的全部信息。元数据自动提供COM中IDL的功能，允许将一个文件同时用于定义和实现。运行库模块和程序集甚至不需要向操作系统注册。运行库使用的说明始终反映编译文件中的实际代码，从而提高应用程序的可靠性。

2．语言互用性和更简单的基于组件的设计

元数据提供所有必需的有关已编译代码的信息，以供您从用不同语言编写的 PE 文件中继承类。您可以创建用任何托管语言（任何面向公共语言运行库的语言）编写的任何类的实例，而不用担心显式封送处理或使用自定义的互用代码。

3．Attribute

.NET Framework允许在编译文件中声明特定种类的元数据（称为Attribute）。在整个 .NET Framework 中到处都可以发现Attribute的存在，Attribute用于更精确地控制运行时程序如何工作。另外，用户可以通过自定义属性向 .NET Framework 文件发出用户自己的自定义元数据。

4.2.1.2元数据的结构

在PE文件中与元数据有关的主要包括两部分。一部分是元数据，它包含一系列的表和堆数据结构。每个元数据表都保留有关程序元素的信息。例如，一个元数据表说明代码中的类，另一个元数据表说明字段等。如果您的代码中有10个类，类表将有10行，每行为1个类。元数据表引用其他的表和堆。例如，类的元数据表引用方法表。元数据以四种堆结构存储信息：字符串、Blob、用户字符串和 GUID。所有用于对类型和成员进行命名的字符串都存储在字符串堆中。例如，方法表不直接存储特定方法的名称，而是指向存储在字符串堆中的方法的名称。

另一部分是MSIL指令，许多MSIL指令都带有元数据标记。元数据标记在 PE 文件的 MSIL 部分中唯一确定每个元数据表的每一行。元数据标记在概念上和指针相似，永久驻留在MSIL中，引用特定的元数据表。元数据标记是一个四个字节的数字。最高位字节表示特定标记（方法、类型等）引用的元数据表。剩下的三个字节指定与所说明的编程元素对应的元数据表中的行。如果用C#定义一个方法并将其编译到PE文件中，下面的元数据标记可能存在于PE文件的MSIL部分：  
0x06000004

最高位字节 (0x06) 表示这是一个MethodDef标记。低位的三个字节 (000004) 指示公共语言运行库在 MethodDef 表的第四行查找对该方法定义进行描述的信息。

表4.1 描述了PE文件中元数据的结构及其每部分的内容：

|  |  |
| --- | --- |
| PE部分 | PE部分的内容 |

表4.1  PE文件中的元数据

4.2.1.3元数据在运行时的作用  
由于在MSIL指令中包含了元数据标记，因此，当公共语言运行库（CLR）将代码加载到内存时，将向元数据咨询该代码模块中包含的信息。运行库对Microsoft 中间语言 (MSIL) 流执行广泛的分析，将其转换为快速本机指令。运行库根据需要使用实时 (JIT) 编译器将 MSIL 指令转换为本机代码，每次转换一个方法。例如，有一个类APP，其中包含了Main()方法和Add()方法：  
using System;

public class App  
{  
   public static int Main()  
   {  
      int ValueOne = 10;  
      int ValueTwo = 20;         
      Console.WriteLine("The Value is: {0}", Add(ValueOne, ValueTwo));  
      return 0;  
   }  
   public static int Add(int One, int Two)  
   {  
      return (One + Two);  
   }  
}

通过运行库，这段代码被加载到内存中，并被转化为MSIL：  
.entrypoint  
.maxstack  3  
.locals ([0] int32 ValueOne,  
         [1] int32 ValueTwo,  
         [2] int32 V\_2,  
         [3] int32 V\_3)  
IL\_0000:  ldc.i4.s   10  
IL\_0002:  stloc.0  
IL\_0003:  ldc.i4.s   20  
IL\_0005:  stloc.1  
IL\_0006:  ldstr      "The Value is: {0}"  
IL\_000b:  ldloc.0  
IL\_000c:  ldloc.1  
IL\_000d:  call int32 ConsoleApplication.MyApp::Add(int32,int32) /\* 06000003 \*/

JIT 编译器读取整个方法的 MSIL，对其进行彻底地分析，然后为该方法生成有效的本机指令。在 IL\_000d 遇到 Add 方法 (/\* 06000003 \*/) 的元数据标记，运行库使用该标记参考 MethodDef 表的第三行。

表4.2显示了说明 Add 方法的元数据标记所引用的 MethodDef 表的一部分：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 行 | 相对虚拟地址 (RVA) | ImplFlags | Flags | Name      （指向字符串堆） | Signature      （指向 Blob 堆） |

表4.2 元数据标记

该表的每一列都包含有关代码的重要信息。RVA 列允许运行库计算定义该方法的 MSIL 的起始内存地址。ImplFlags 和 Flags 列包含说明该方法的位屏蔽（例如，该方法是公共的还是私有的）。Name 列对来自字符串堆的方法的名称进行了索引。Signature 列对在 Blob 堆中的方法签名的定义进行了索引。

通过利用元数据，我们就可以获得类的相关信息。如上所述，在类APP的MethodDef表中，可以获得类APP的三个方法，以及方法的Flags和方法签名。而在.Net中，则提供了反射技术，来支持这种对元数据信息的获取。可以说，正是因为有了元数据，才使得AOP的拦截与织入功能的实现成为可能。

4.2.2 Attribute  
4.2.2.1 Attribute概述

通过对.Net元数据的分析，我们知道可以通过Attribute来扩展元数据。那么什么是Attribute？在MSDN中，Attribute被定义为“是被指定给某一声明的一则附加的声明性信息”。 我们可以通过Attribute来定义设计层面的信息以及运行时（run-time）信息，也可以利用Attribute建立自描述（self-describing）组件。

Attribute可应用于任何目标元素，我们可以通过AttributeTargets枚举指定其施加的目标，AttributeTargets枚举在.Net中的定义如下：  
public enum AttributeTargets  
{  
   All=16383,  
   Assembly=1,  
   Module=2,  
   Class=4,  
   Struct=8,  
   Enum=16,  
   Constructor=32,  
   Method=64,  
   Property=128,  
   Field=256,  
   Event=512,  
   Interface=1024,  
   Parameter=2048,  
   Delegate=4096,  
   ReturnValue=8192  
}

作为参数的AttributeTarges的值允许通过“或”操作来进行多个值的组合，如果你没有指定参数，那么默认参数就是All 。

不管是.Net Framework提供的Attribute，还是用户自定义Attribute，都是通过[]施加到目标元素上。虽然Attribute的用法与通常的类型不一样，但在.Net内部，Attribute本质上还是一个类。但是，Attribute类的实例化发生在编译时，而非运行时，因而达到了扩展元数据的目的。一个Attribute的多个实例可应用于同一个目标元素；并且Attribute可由从目标元素派生的元素继承。

4.2.2.2自定义Attribute  
.Net Framework支持用户自定义Attribute。自定义Attribute的方法与定义类一样，唯一不同之处是自定义的Attribute必须继承Attribute类。Attribute类包含用于访问和测试自定义Attribute的简便方法。其中，Attribute类的构造函数为protected，只能被Attribute的派生类调用。Attribute类包含的方法主要为：

1．三个静态方法  
static Attribute GetCustomAttribute():这个方法有8种重载的版本，它被用来取出施加在类成员上指定类型的Attribute。  
static Attribute[] GetCustomAttributes(): 这个方法有16种重载版本，用来取出施加在类成员上指定类型的Attribute数组。  
static bool IsDefined()：有八种重载版本，看是否指定类型的定制attribute被施加到类的成员上面。

2．两个实例方法  
bool IsDefaultAttribute(): 如果Attribute的值是默认的值，那么返回true。  
bool Match():表明这个Attribute实例是否等于一个指定的对象。

3．公共属性  
TypeId: 得到一个唯一的标识，这个标识被用来区分同一个Attribute的不同实例。

通过自定义Attribute，可使得用户自定义的信息与Attribute施加的类本身相关联。例如，给定一个自定义的 .NET 属性，我们就可以轻松地将调用跟踪Attribute与类的方法相关联：  
public class Bar  
{  
    [CallTracingAttribute("In Bar ctor")]  
    public Bar() {}  
    [CallTracingAttribute("In Bar.Calculate method")]  
    public int Calculate(int x, int y){ return x + y; }  
}

请注意，方括号中包含 CallTracingAttribute 和访问方法时输出的字符串。这是将自定义元数据与 Bar 的两个方法相关联的Attribute语法。该自定义的Attribute实现，如下所示：  
using System;  
using System.Reflection;

[AttributeUsage( AttributeTargets.ClassMembers, AllowMultiple = false )]  
public class CallTracingAttribute : Attribute  
{      
    private string m\_TracingInfo;  
    public CallTracingAttribute(string info)  
    {  
        m\_TracingInfo = info;  
    }  
    public string TracingInfo  
    {  
        get {return tracingInfo;}  
    }  
}

通过自定义的CallTracingAttribute，将一段Tracing信息施加到类Bar的构造函数和方法Calculate上。我们可以利用反射技术与Attribute类提供的方法，来获得Bar类的元数据中包含的Attribute信息，如：  
public class Test  
{  
    public static void Main(string[] args)  
    {  
        System.Reflection.MemberInfo info = typeof(Bar);  
        CallTracingAttribute attribute = (CallTracingAttribute) Attribute.GetCustomAttribute(info,typeof(CallTracingAttribute));  
        if (attribute != null)  
        {  
             Console.WriteLine(“Tracing Information:{0}”,attribute.TracingInfo);  
        }  
    }  
}

4.2.2.3上下文（Context）和Attribute  
所谓上下文（Context），是指一个逻辑上的执行环境。每一个应用程序域都有一个或多个Context，.Net中的所有对象都会在相应的Context中创建和运行。如图4.1所示，它显示了一个安全地存在于Context的对象：



图4.1 安全地存在于Context的对象

在图4.1中，上下文（Context）提供了错误传播、事务管理和同步功能，而对象的创建和运行就存在于该Context中。在.Net中，提供了ContextBoundObject类，它代表的含义就是该对象应存在于指定的Context边界中（Object that will be bound with a context）。凡是继承了ContextBoundObject类的类类型，就自动具备了对象与Context之间的关系。事实上，如果一个类对象没有继承自ContextBoundObject，则该对象默认会创建和运行在应用程序域的default context中，而继承自ContextBoundObject的类对象，在其对象实例被激活时，CLR将自动创建一个单独的Context供其生存。

如果需要判定ContextBoundObject类型对象所认定的Context，只需要为该类型对象施加ContextAttribute即可。ContextAttribute类继承了Attribute类，它是一个特殊的Attribute，通过它，可以获得对象需要的合适的执行环境，即Context（上下文）。同时，ContextAttribute还实现了IContextAttribute和IContextProperty接口。

由于在施加Attribute时，只需要获取ContextBoundObject类型的Context属性，因此，我们也可以自定义Attribute，只需要该自定义的Attribute实现IContextAttribute即可。IContextAttribute接口的定义如下：  
public interface IContextAttribute  
{  
    bool IsContextOK(Context ctx, IConstructionCallMessage ctorMsg);  
    void GetPropertiesForNewContext(IConstructionCallMessage ctorMsg);  
}

每个context attribute在context的构造阶段（通常是由ContextBoundObject对象构造动作引发的）会被首先问到IsContextOK，就是说新创建的这个ContextBoundObjec（通过ctorMsg可以知道是哪个对象的哪个构造方法被用来构造ContextBoundObjec对象的）能不能在给定的ctx中存在？这个目的主要是减少应用程序域中潜在的context的数量，如果某些ContextBoundObjec类型可以共用一个有所需特性的执行环境的话，就可以不用再创建新的环境，而只要在已有的环境中构造并执行就好了。

如果ContextBoundObjec类型上设置的所有context attributes都认同给定的context（也即调用代码所处的context）是正确地的（此时IsContextOK均返回true），那么新的ContextBoundObjec就会被绑定到这个context上。否则，只有有一个attribute返回false，就会立即创建一个新的context。然后，CLR会再一次询问每一个context attribute新构造的context是否正确，由于Context已经被重新创建，通常此时返回的结果应为false。那么，Context构造程序就会调用其GetPropertiesForNewContext()方法，context attribute可以用这个方法传入的构造器方法调用信息（ctorMsg）中的context properties列表（ContextProperties）来为新建的context增加所需的context properties。

从AOP的角度来看，Context类似于前面分析的横切关注点，那么利用我们自定义的Context Attribute，就可以获得对象它所存在的上下文，从而建立业务对象与横切关注点之间的关系。

4.2.3代理（Proxy）  
在程序设计中使用代理（Proxy），最重要的目的是可以通过利用代理对象，实现代理所指向的真实对象的访问。在GOF的《设计模式》中，将代理（Proxy）模式分为四种：  
1、远程代理（Remote Proxy）。它为一个位于不同的地址空间的对象提供一个局域代表对象。这个不同的地址空间可以是在本机器中，亦可是在另一台机器中。  
2、虚代理（Virtual Proxy）。它能够根据需要创建一个资源消耗较大的对象，使得此对象只在需要时才会被真正创建。  
3、保护代理（Protection Proxy）。它控制对原始对象的访问，如果需要可以给不同的用户提供不同级别的使用权限。  
4、智能引用代理（Smart Reference Proxy）。它取代了简单的指针，在访问一个对象时，提供一些额外的操作。例如，对指向实际对象的引用计数，这样当该对象没有引用时，可以自动释放它。当第一次引用一个持久对象时，智能引用可以将该对象装入内存。在访问一个实际对象前，检查该对象是否被锁定，以确保其他对象不能改变它。

在.Net Remoting中，采用了远程代理（Remote Proxy）模式。采用代理技术，使得对象可以在两个不同的应用程序域（甚至可以是两台不同的机器）之间传递。代理在.Net中被分为透明代理（Transparent Proxy）和真实代理（Real Proxy）。Transparent Proxy的目标是在 CLR 中在 IL 层面最大程度扮演被代理的远端对象，从类型转换到类型获取，从字段访问到方法调用。对 CLR 的使用者来说，Transparent Proxy和被其代理的对象完全没有任何区别，只有通过 RemotingServices.IsTransparentProxy 才能区分两者的区别。Real Proxy则是提供给 CLR 使用者扩展代理机制的切入点，通过从Real Proxy继承并实现 Invoke 方法，用户自定义代理实现可以自由的处理已经被从栈调用转换为消息调用的目标对象方法调用，如实现缓存、身份验证、安全检测、延迟加载等等。

如果我们希望自己定义的代理类能够“模仿”真实对象的能力，首先就需要实现透明代理。然而，CLR中虽然提供了这样一个透明代理类（\_TransparentProxy），我们却不能让自己的代理类从透明代理类派生，也不能通过自定义Attribute、实现标志性接口等方式将代理类标识为透明代理，从而让CLR能够认识。要获取透明代理，必须要提供一个真实代理。一个真实代理是一个从System.Runtime.Remoting.Proxies.RealProxy派生而来的类。这个RealProxy类的首要功能就是帮我们在运行期动态生成一个可以透明兼容于某一个指定类的透明代理类实例。从RealProxy的源代码，可以看出透明代理和真实代理之间的关系：  
namespace System.Runtime.Remoting.Proxies  
{  
  abstract public class RealProxy  
  {  
    protected RealProxy(Type classToProxy) : this(classToProxy, (IntPtr)0, null){}  
    protected RealProxy(Type classToProxy, IntPtr stub, Object stubData)  
    {  
      if(!classToProxy.IsMarshalByRef && !classToProxy.IsInterface)  
        throw new ArgumentException(...);

      if((IntPtr)0 == stub)  
      {  
        stub = \_defaultStub;  
        stubData = \_defaultStubData;  
      }

      \_tp = null;

      if (stubData == null)  
        throw new ArgumentNullException("stubdata");

      \_tp = RemotingServices.CreateTransparentProxy(this, classToProxy, stub, stubData);  
    }  
    public virtual Object GetTransparentProxy()  
    {  
      return \_tp;  
    }  
  }  
}

很明显，透明代理（Transparent Proxy）是在RealProxy类的构造函数中，调用RemotingServices.CreateTransparentProxy()方法动态创建的。CreateTransparentProxy()方法将把被代理的类型强制转换为统一的由 CLR 在运行时创建的 RuntimeType 类型，进而调用 Internal 方法完成TransparentProxy的创建。通过GetTransparentProxy()方法，就可以获得创建的这个透明代理对象。因此，要定义自己的真实代理对象，只需要继承RealProxy类即可：  
using System.Runtime.Remoting.Proxies;

public class MyRealProxy: RealProxy  
{  
  public MyRealProxy(Type classToProxy): base(classToProxy)  
  {  
    …  
  }  
}

透明代理和真实代理在上下文（Context）中，会起到一个侦听器的作用。首先，透明代理将调用堆栈序列化为一个称为消息（Message）的对象，然后再将消息传递给真实代理。真实代理接收消息，并将其发送给第一个消息接收进行处理。第一个消息接收对消息进行预处理，将其继续发送给位于客户端和对象之间的消息接收堆栈中的下一个消息接收，然后对消息进行后处理。下一个消息接收也如此照办，以此类推，直到到达堆栈构建器接收，它将消息反序列化回调用堆栈，调用对象，序列化出站参数和返回值，并返回到前面的消息接收。这个调用链如图4.2所示。

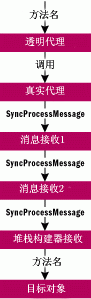


图4.2 代理（Proxy）侦听消息的顺序

由于透明代理完全等同于其代理的对象，因此，当我们侦听到代理对象被调用的消息时，就可以截取该消息，并织入需要执行的方面逻辑，完成横切关注逻辑与核心逻辑的动态代码织入。

4.3 .Net平台下AOP技术实现  
4.3.1实现原理  
根据对.Net中元数据（Metadata）、Attribute、上下文（Context）、代理（Proxy）等技术要素的分析，要在.Net中实现AOP，首先需要获得一个类对象的上下文（Context），则其前提就是这个类必须从System.ContextBoundObject类派生。这个类对象就相当于AOP中的核心关注点，而类对象的上下文则属于AOP的横切关注点。很显然，只需要利用上下文，就可以方便的实现核心关注点和横切关注点的分离。

正如图4.1所示，对象是存在于上下文中的。利用自定义Attribute，可以建立对象与上下文之间的关联。Attribute可以扩展对象的元数据，从而标识出该对象属于其中的一个或多个Aspect。一旦该对象实例被创建或调用时，就可以利用反射技术获得该对象的自定义Attribute。为使得对象的元数据与上下文关联起来，就要求这个自定义的Attribute必须实现接口IContextAttribute。

获得了对象的上下文之后，透明代理与真实代理就能够对该对象的方法调用（包括构造函数）进行侦听，并完成消息的传递。传递的消息可以被Aspect截取，同时利用真实代理，也可以完成对业务对象的Decorate，将Aspect逻辑注入到业务对象中。由于在大型的企业系统设计中，横切关注点会包括事务管理、日志管理、权限控制等多方面，但由于方面（Aspect）在技术上的共同特性，我们可以利用.Net的相关技术实现方面（Aspect）的核心类库，所有的横切关注点逻辑，都可以定义为派生这些类库的类型，从而真正在.Net中实现AOP技术。

4.3.2 AOP公共类库  
4.3.2.1 AOP Attribute  
如上所述，要实现AOP技术，首先需要自定义一个Attribute。该自定义Attribute必须实现IContextAttribute，因此其定义如下所示：  
using System;  
using System.Runtime.Remoting.Contexts;  
using System.Runtime.Remoting.Activation;

[AttributeUsage(AttributeTargets.Class)]  
public abstract class AOPAttribute:Attribute,IContextAttribute  
{  
    private string m\_AspectXml;  
    private const string CONFIGFILE = @"configuration\aspect.xml";  
    public AOPAttribute()                      
    {  
        m\_AspectXml = CONFIGFILE;  
    }     
    public AOPAttribute(string aspectXml)  
    {  
        this.m\_AspectXml = aspectXml;  
    }    
    protected abstract AOPProperty GetAOPProperty();

    #region IContextAttribute Members  
    public sealed void GetPropertiesForNewContext(IConstructionCallMessage ctorMsg)  
    {  
        AOPProperty property = GetAOPProperty();      
        property.AspectXml = m\_AspectXml;       
        ctorMsg.ContextProperties.Add(property);  
    }  
    public bool IsContextOK(Context ctx, IConstructionCallMessage ctorMsg)  
    {  
        return false;  
    }  
}

类AOPAttribute除了继承System.Attribute类之外，关键之处在于实现了接口IContextAttribute接口。接口方法GetPropertiesForNewContext()其功能是向Context添加属性（Property）集合，这个集合是IConstructionCallMessage对象的ContextProperties属性。而接口方法IsContextOK()，则用于判断Context中是否存在指定的属性。这个方法会在Context的构造阶段（通常是由被施加了AOPAttribute的业务对象在创建时引发的）被调用，如果返回false，会创建一个新的Context。

GetAOPProperty()方法是一个受保护的抽象方法，继承AOPAttribute的子类将重写该方法，返回一个AOPProperty对象。在这里，我们利用了Template Method模式，通过该方法创建符合条件的AOPProperty对象，并被GetPropertiesForNewContext()方法添加到属性集合中。

抽象类AOPAttribute是所有与方面有关的Attribute的公共基类。所有方面的相关Attribute均继承自它，同时实现GetAOPProperty()方法，创建并返回与之对应的AOPProperty对象。

4.3.2.2 AOP Property  
ContextProperties是一个特殊的集合对象，它存放的是对象被称为Context Property，是一个实现了IContextProperty接口的对象，这个对象可以为相关的Context提供一些属性。IContextProperty接口的定义如下：  
public interface IContextProperty  
{  
    string Name { get; }  
    bool IsNewContextOK(Context newCtx);  
    void Freeze(Context newCtx);  
}

IContextProperty接口的Name属性，表示Context Property的名字，Name属性值要求在整个Context中必须是唯一的。IsNewContextOK()方法用于确认Context是否存在冲突的情况。而Freeze()方法则是通知Context Property，当新的Context构造完成时，则进入Freeze状态（通常情况下，Freeze方法仅提供一个空的实现）。

由于IContextProperty接口仅仅是为Context提供一些基本信息，它并不能完成对方法调用消息的截取。根据对代理技术的分析，要实现AOP，必须在方法调用截取消息传递，并形成一个消息链Message Sink。因此，如果需要向所在的Context的Transparent Proxy/Real Proxy中植入Message Sink，Context Property还需要提供Sink的功能。所幸的是，.Net已经提供了实现MessageSink功能的相关接口，这些接口的命名规则为IContributeXXXSink，XXX代表了四种不同的Sink：Envoy，ClientContext，ServerContext，Object。这四种接口有其相似之处，都只具有一个方法用于返回一个IMessageSink对象。由于我们需要获取的透明代理对象，是能够穿越不同的应用程序域的。在一个应用程序域收到其他应用程序域的对象，则该对象在.Net中被称为Server Object，该对象所处的Context也被称为Server Context。我们在.Net中实现AOP，其本质正是要获得对象的Server Context，并截取该Context中的方法调用消息，因而Context Property对象应该实现IContributeServerContextSink接口。事实上，也只有IContributeServerContextSink接口的GetServerContextSink()方法，才能拦截包括构造函数在内的所有方法的调用。

因此，AOP Property最终的定义如下：  
using System;  
using System.Runtime.Remoting.Activation;  
using System.Runtime.Remoting.Contexts;  
using System.Runtime.Remoting.Messaging;

public abstract class AOPProperty : IContextProperty, IContributeServerContextSink  
{  
    private string m\_AspectXml;  
    public AOPProperty()  
    {  
        m\_AspectXml = string.Empty;            
    }  
    public string AspectXml  
    {  
        set { m\_AspectXml = value; }  
    }  
    protected abstract IMessageSink CreateAspect(IMessageSink nextSink);  
    protected virtual string GetName()  
    {  
        return "AOP";  
    }  
    protected virtual void FreezeImpl(Context newContext)  
    {  
        return;  
    }  
    protected virtual bool CheckNewContext(Context newCtx)  
    {  
        return true;  
    }

    #region IContextProperty Members  
    public void Freeze(Context newContext)  
    {  
        FreezeImpl(newContext);  
    }  
    public bool IsNewContextOK(Context newCtx)  
    {  
        return CheckNewContext(newCtx);  
    }  
    public string Name  
    {  
        get { return GetName(); }  
    }  
    #endregion

    #region IContributeServerContextSink Members  
    public IMessageSink GetServerContextSink(IMessageSink nextSink)  
    {  
        Aspect aspect = (Aspect)CreateAspect(nextSink);             
        aspect.ReadAspect(m\_AspectXml,Name);             
        return (IMessageSink)aspect;  
    }  
    #endregion  
}

在抽象类AOPProperty中，同样利用了Template Method模式，将接口IContextProperty的方法的实现利用受保护的虚方法延迟到继承AOPProperty的子类中。同时，对于接口IContributeServerContextSink方法GetServerContextSink()，则创建并返回了一个Aspect类型的对象，Aspect类型实现了IMessageSink接口，它即为AOP中的方面，是所有方面（Aspect）的公共基类。

AOPProperty类作为抽象类，是所有与上下文有关的Property的公共基类。作为Context Property应与Aspect相对应，且具体的AOPProperty类对象应在AOPAttribute的子类中创建并获得。

4.3.2.3 Aspect与PointCut  
Aspect类是AOP的核心，它的本质是一个Message Sink，代理正是通过它进行消息的传递，并截获方法间传递的消息。Aspect类实现了IMessageSink接口，其定义如下：  
public interface IMessageSink  
{  
    IMessage SyncProcessMessage(IMessage msg);  
    IMessageCtrl AsyncProcessMessage(IMessage msg, IMessageSink replySink);  
    IMessageSink NextSink { get; }      
}

IMessageSink接口利用NextSink将多个Message Sink连接起来，以形成一个消息接收器链；而SyncProcessMessage()和AsyncProcessMessage()方法则分别用于同步和异步操作，它们在消息传递的时候被调用。

注意方法SyncProcessMessage()中的参数，是一个IMessage接口类型的对象。在.Net中，IMethodCallMessage和IMethodReturnMessage接口均继承自IMessage接口，前者是调用方法的消息，而后者则是方法被调用后的返回消息。利用这两个接口对象，就可以获得一个对象方法的切入点。因此，一个最简单的Aspect实现应该如下：  
    public abstract class Aspect : IMessageSink  
    {       
        private IMessageSink m\_NextSink;

        public AOPSink(IMessageSink nextSink)  
        {  
            m\_NextSink = nextSink;          
        }  
        public IMessageSink NextSink  
        {  
            get { return m\_NextSink; }  
        }  
        public IMessage SyncProcessMessage(IMessage msg)  
        {  
            IMethodCallMessage call = msg as IMethodCallMessage;  
            if (call == null)  
            {  
                 return null;  
            }

            IMessage retMsg = null;  
            BeforeProcess();  
            retMsg = m\_NextSink.SyncProcessMessage(msg);  
            AfterProcess();  
            return retMsg;  
        }  
        public IMessageCtrl AsyncProcessMessage(IMessage msg, IMessageSink replySink)  
        {  
             return null;  
        }  
        private void BeforeProcess()  
        {  
             //方法调用前的实现逻辑；  
        }  
        private void AfterProcess()  
        {  
             //方法调用后的实现逻辑；  
        }  
    }

注意在方法SyncProcessMessage()中，IMessageSink对象m\_NextSink通过Aspect构造函数赋值为业务对象的透明代理，在调用m\_NextSink的SyncProcessMessage()方法时，此时调用的是与该透明代理对应的真实代理。如果在消息接收链中还存在代理，在方法调用将会沿着消息链不断的向后执行。而对于一个业务对象而言，此时的IMessage即为该对象中被调用的方法，SyncProcessMessage(msg)就相当于执行了该方法。而在m\_NextSink.SyncProcessMessage(msg)方法前后执行BeforeProcess()和AfterProcess()，就完成了对方法的截取，并将自己的Aspect逻辑织入到业务对象的方法调用中，从而实现了AOP。

然而对于AOP技术的实际应用而言，并非业务对象的所有方法都需要被截取进而进行方面的织入。也即是说，切入点（PointCut）必须实现可被用户定义。而所谓切入点，实际上是业务对象方法与Advice之间的映射关系。在.Net中，我们可以通过集合对象来管理这个映射关系。由于Advice包括Before Advice和After Advice，因此，在Aspect类中应该定义两个集合对象：  
private SortedList m\_BeforeAdvices;  
private SortedList m\_AfterAdvices;

在添加PointCut时，是将方法名和具体的Advice对象建立映射，根据SortedList集合的特性，我们将方法名作为SortedList的Key，而Advice则作为SortedList的Value：  
        protected virtual void AddBeforeAdvice(string methodName, IBeforeAdvice before)  
        {  
            lock (this.m\_BeforeAdvices)  
            {  
                if (!m\_BeforeAdvices.Contains(methodName))  
                {  
                    m\_BeforeAdvices.Add(methodName, before);  
                }  
            }  
        }  
        protected virtual void AddAfterAdvice(string methodName, IAfterAdvice after)  
        {  
            lock (this.m\_AfterAdvices)  
            {  
                if (!m\_AfterAdvices.Contains(methodName))  
                {  
                    m\_AfterAdvices.Add(methodName, after);  
                }  
            }  
        }

在向SortedList添加PointCut时，需要先判断集合中是否已经存在该PointCut。同时考虑到可能存在并发处理的情况，在添加PointCut时，利用lock对该操作进行了加锁，避免并发处理时可能会出现的错误。

建立了方法名和Advice的映射关系，在执行SyncProcessMessage()方法，就可以根据IMessage的值，获得业务对象被调用方法的相关属性，然后根据方法名，找到其对应的Advice，从而执行相关的Advice代码：  
       public IMessage SyncProcessMessage(IMessage msg)  
       {             
             IMethodCallMessage call = msg as IMethodCallMessage;  
             string methodName = call.MethodName.ToUpper();  
             IBeforeAdvice before = FindBeforeAdvice(methodName);  
             if (before != null)  
             {  
                  before.BeforeAdvice(call);  
             }             
             IMessage retMsg = m\_NextSink.SyncProcessMessage(msg);  
             IMethodReturnMessage reply = retMsg as IMethodReturnMessage;  
             IAfterAdvice after = FindAfterAdvice(methodName);  
             if (after != null)  
             {  
                   after.AfterAdvice(reply);  
             }  
             return retMsg;  
        }

其中FindBeforeAdvice()和FindAfterAdvice()方法完成key和value的查找工作，分别的定义如下：  
        public IBeforeAdvice FindBeforeAdvice(string methodName)  
        {  
            IBeforeAdvice before;  
            lock (this.m\_BeforeAdvices)  
            {  
                before = (IBeforeAdvice)m\_BeforeAdvices[methodName];  
            }  
            return before;  
        }  
        public IAfterAdvice FindAfterAdvice(string methodName)  
        {  
            IAfterAdvice after;  
            lock (this.m\_AfterAdvices)  
            {  
                after = (IAfterAdvice)m\_AfterAdvices[methodName];  
            }  
            return after;  
        }

在找到对应的Advice对象后，就可以调用Advice对象的相关方法，完成方面逻辑代码的织入。

那么，PointCut是在什么时候添加的呢？我们可以在AOP的配置文件（Aspect.xml）中配置PointCut，然后在Aspect类中，通过ReadAspect()方法，读入配置文件，获取PointCut以及Aspect需要的信息，包括方法名和Advice对象（通过反射动态创建），在执行AddBeforeAdvice()和AddAfterAdvice()方法将PointCut添加到各自的集合对象中：  
public void ReadAspect(string aspectXml,string aspectName)  
{  
    IBeforeAdvice before = (IBeforeAdvice)Configuration.GetAdvice(aspectXml,aspectName,Advice.Before);  
    string[] methodNames = Configuration.GetNames(aspectXml,aspectName,Advice.Before);  
    foreach (string name in methodNames)  
    {  
         AddBeforeAdvice(name,before);  
    }  
    IAfterAdvice after = (IAfterAdvice)Configuration.GetAdvice(aspectXml,aspectName,Advice.After);  
    string[] methodNames = Configuration.GetNames(aspectXml,aspectName,Advice.After);  
    foreach (string name in methodNames)  
    {  
         AddAfterAdvice(name,after);  
    }     
}

一个Aspect的配置文件示例如下：  
<aop>  
    <aspect value ="LogAOP">  
        <advice type="before" assembly="AOP.Advice" class="AOP.Advice.LogAdvice">  
     <pointcut>ADD</pointcut>  
     <pointcut>SUBSTRACT</pointcut>  
 </advice>  
 <advice type="after" assembly="AOP.Advice" class="AOP.Advice.LogAdvice">  
     <pointcut>ADD</pointcut>  
     <pointcut>SUBSTRACT</pointcut>  
 </advice>  
    </aspect>   
</aop>

配置文件中，元素Advice的assembly属性和class属性值，是利用反射创建Advice对象所需要的信息。另外，Aspect的名字应与方面的Property名保持一致，因为ReadAspect()方法是通过AOPProperty名字来定位配置文件的Aspect。

4.3.2.4 Advice  
在Aspect类中，已经使用了Advice对象。根据类别不同，这些Advice对象分别实现IBeforeAdvice接口和IAfterAdvice接口：  
using System;  
using System.Runtime.Remoting.Messaging;

public interface IBeforeAdvice  
{  
    void BeforeAdvice(IMethodCallMessage callMsg);  
}  
public interface IAfterAdvice  
{  
    void AfterAdvice(IMethodReturnMessage returnMsg);  
}

接口方法应该实现具体的方面逻辑，同时可以通过IMethodCallMessage对象获得业务对象的调用方法信息，通过IMethodReturnMessage对象获得方法的返回信息。

4.4 .Net平台AOP技术应用案例  
在4.3.2节，我们已基本实现了AOP的公共类库，这其中包括AOPAttribute，AOPProperty，Aspect，IBeforeAdvice，IAfterAdvice。根据这些公共基类或接口，我们就可以定义具体的方面，分别继承或实现这些类与接口。为了展示AOP在.Net中的应用，在本节，我将以一个简单的实例来说明。

假定我们要设计一个计算器，它能提供加法和减法功能。我们希望，在计算过程中，能够通过日志记录整个计算过程及其结果，同时需要监测其运算性能。该例中，核心业务是加法和减法，而公共的业务则是日志与监测功能。根据前面对AOP的分析，这两个功能作为横切关注点，将是整个系统需要剥离出来的“方面”。

4.4.1日志方面  
     作为日志方面，其功能就是要截取业务对象方法的调用，并获取之间传递的消息内容。从上节的分析我们知道，方法间的消息可以从IMethodCallMessage和IMethodReturnMessage接口对象获得。因此，实现日志方面，最重要的是实现Aspect类中的SyncProcessMessage()方法。此外，也应定义与之对应的Attribute和Property，以及实现日志逻辑的Advice。

4.4.1.1日志Attribute（LogAOPAttribute）  
LogAOPAttribute类继承AOPAttribute，由于AOPAttribute类主要是创建并获得对应的AOPProperty，因此，其子类也仅需要重写父类的受保护抽象方法GetAOPProperty()即可：  
       [AttributeUsage(AttributeTargets.Class)]  
       public class LogAOPAttribute:AOPAttribute  
       {  
              public LogAOPAttribute():base()  
              {}  
              public LogAOPAttribute(string aspectXml):base(aspectXml)  
              {}

              protected override AOPProperty GetAOPProperty()  
              {  
                     return new LogAOPProperty();  
              }     
       }

通过对GetAOPProperty()方法的重写，创建并获得了与LogAOPAttribute类相对应的LogAOPProperty，此时在LogAOPAttribute所施加的业务对象的上下文中，所存在的AOP Property就应该是具体的LogAOPProperty对象。

4.4.1.2日志Property（LogAOPProperty）  
由于Context Property的名字在上下文中必须是唯一的，因此每个方面的Property的名字也必须是唯一的。因此在继承AOPProperty的子类LogAOPProperty中，必须重写父类的虚方法GetName()，同时在LogAOPProperty中，还应该创建与之对应的Aspect，也即是Message Sink，而这个工作是由抽象方法CreateAspect()来完成的。因此，LogAOPProperty类的定义如下：  
       public class LogAOPProperty:AOPProperty  
       {  
              protected override IMessageSink CreateAspect(IMessageSink nextSink)  
              {  
                     return new LogAspect(nextSink);  
              }  
              protected override string GetName()  
              {  
                     return "LogAOP";  
              }  
       }

为避免Property的名字出现重复，约定成俗以方面的Attribute名为Property的名字，以本例而言，其Property名为LogAOP。

4.4.1.3日志Aspect（LogAspect）  
LogAspect完成的功能主要是将Advice与业务对象的方法建立映射，并将其添加到Advice集合中。由于我们在AOP实现中，利用了xml配置文件来配置PointCut，因此对于所有Aspect而言，这些操作都是相同的，只要定义了正确的配置文件，将其读入即可。对于Aspect的SyncProcessMessage()，由于拦截和织入的方法是一样的，不同的只是Advice的逻辑而已，因此在所有Aspect的公共基类中已经提供了默认的实现：  
       public class LogAspect:Aspect  
       {  
              public LogAspect(IMessageSink nextSink):base(nextSink)  
              {}             
       }

然后定义正确的配置文件：  
<aspect value ="LogAOP">  
    <advice type="before" assembly=" AOP.Advice" class="AOP.Advice.LogAdvice">  
        <pointcut>ADD</pointcut>  
 <pointcut>SUBSTRACT</pointcut>  
    </advice>  
    <advice type="after" assembly=" AOP.Advice" class="AOP.Advice.LogAdvice">  
 <pointcut>ADD</pointcut>  
 <pointcut>SUBSTRACT</pointcut>  
    </advice>  
</aspect>

LogAdvice所属的程序集文件为AOP.Advice.dll，完整的类名为AOP.Advice.LogAdvice。

4.4.1.4日志Advice（LogAdvice）  
由于日志方面需要记录方法调用前后的相关数据，因此LogAdvice应同时实现IBeforeAdvice和IAfterAdvice接口：  
    public class LogAdvice:IAfterAdvice,IBeforeAdvice  
    {  
        #region IBeforeAdvice Members  
        public void BeforeAdvice(IMethodCallMessage callMsg)  
        {  
            if (callMsg == null)  
            {  
                return;  
            }  
            Console.WriteLine("{0}({1},{2})", callMsg.MethodName, callMsg.GetArg(0), callMsg.GetArg(1));  
        }  
        #endregion

        #region IAfterAdvice Members  
        public void AfterAdvice(IMethodReturnMessage returnMsg)  
        {  
            if (returnMsg == null)  
            {  
                return;  
            }  
            Console.WriteLine("Result is {0}", returnMsg.ReturnValue);  
        }  
        #endregion  
    }

在BeforeAdvice()方法中，消息类型为IMethodCallMessage，通过这个接口对象，可以获取方法名和方法调用的参数值。与之相反，AfterAdvice()方法中的消息类型为IMethodReturnMessage，Advice所要获得的数据为方法的返回值ReturnValue。

4.4.2性能监测方面  
性能监测方面与日志方面的实现大致相同，为简便起见，我要实现的性能监测仅仅是记录方法调用前和调用后的时间。

4.4.2.1性能监测Attribute（MonitorAOPAttribute）  
与日志Attribute相同，MonitorAOPAttribute仅仅需要创建并返回对应的MonitorAOPProperty对象：  
      [AttributeUsage(AttributeTargets.Class)]  
      public class MonitorAOPAttribute:AOPAttribute  
      {  
             public MonitorAOPAttribute():base()  
             {}  
             public MonitorAOPAttribute(string aspectXml):base(aspectXml)  
             {}  
             protected override AOPProperty GetAOPProperty()  
             {  
                    return new MonitorAOPProperty();  
             }   
      }

4.4.2.2性能监测Property（MonitorAOPProperty）  
MonitorAOPProperty的属性名将定义为MonitorAOP，使其与日志方面的属性区别。除定义性能监测方面的属性名外，还需要重写CreateAspect()方法，创建并返回对应的方面对象MonitorAspect：  
      public class MonitorAOPProperty:AOPProperty  
      {  
             protected override IMessageSink CreateAspect(IMessageSink nextSink)  
             {  
                    return new MonitorAspect(nextSink);  
             }  
             protected override string GetName()  
             {  
                    return "MonitorAOP";  
             }  
      }

4.4.2.3性能监测Aspect（MonitorAspect）  
MonitorAspect类的实现同样简单：  
    public class MonitorAspect:Aspect  
    {  
             public MonitorAspect(IMessageSink nextSink):base(nextSink)  
             {}  
    }  
  
而其配置文件的定义则如下所示：  
<aspect value ="MonitorAOP">  
    <advice type="before" assembly=" AOP.Advice" class="AOP.Advice.MonitorAdvice">  
        <pointcut>ADD</pointcut>  
 <pointcut>SUBSTRACT</pointcut>  
    </advice>  
    <advice type="after" assembly=" AOP.Advice" class="AOP.Advice.MonitorAdvice">  
 <pointcut>ADD</pointcut>  
 <pointcut>SUBSTRACT</pointcut>  
    </advice>  
</aspect>

MonitorAdvice所属的程序集文件为AOP.Advice.dll，完整的类名为AOP.Advice.MonitorAdvice。

4.4.2.4性能监测Advice（MonitorAdvice）  
由于性能监测方面需要记录方法调用前后的具体时间，因此MonitorAdvice应同时实现IBeforeAdvice和IAfterAdvice接口：  
    public class MonitorAdvice : IBeforeAdvice, IAfterAdvice  
    {  
        #region IBeforeAdvice Members  
        public void BeforeAdvice(IMethodCallMessage callMsg)  
        {  
            if (callMsg == null)  
            {  
                return;  
            }  
            Console.WriteLine("Before {0} at {1}", callMsg.MethodName, DateTime.Now);  
        }  
        #endregion

        #region IAfterAdvice Members  
        public void AfterAdvice(IMethodReturnMessage returnMsg)  
        {  
            if (returnMsg == null)  
            {  
                return;  
            }  
            Console.WriteLine("After {0} at {1}", returnMsg.MethodName, DateTime.Now);  
        }  
        #endregion  
    }

MonitorAdvice只需要记录方法调用前后的时间，因此只需要分别在BeforeAdvice()和AfterAdvice()方法中，记录当前的时间即可。

4.4.3业务对象与应用程序  
4.4.3.1业务对象（Calculator）  
通过AOP技术，我们已经将核心关注点和横切关注点完全分离，我们在定义业务对象时，并不需要关注包括日志、性能监测等方面，这也是AOP技术的优势。当然，由于要利用.Net中的Attribute及代理技术，对于施加了方面的业务对象而言，仍然需要一些小小的限制。

首先，我们应该将定义好的方面Aspect施加给业务对象。其次，由于代理技术要获取业务对象的上下文（Context），该上下文必须是指定的，而非默认的上下文。上下文的获得，是在业务对象创建和调用的时候，如果要获取指定的上下文，在.Net中，要求业务对象必须继承ContextBoundObject类。因此，最后业务对象Calculator类的定义如下所示：  
       [MonitorAOP]  
       [LogAOP]  
       public class Calculator : ContextBoundObject  
       {  
              public int Add(int x,int y)  
              {  
                     return x + y;  
              }  
              public int Substract(int x,int y)  
              {  
                     return x - y;  
              }  
       }

[MonitorAOP]和[LogAOP]正是之前定义的方面Attribute，此外Calculator类继承了ContextBoundObject。除此之外，Calculator类的定义与普通的对象定义无异。然而，正是利用AOP技术，就可以拦截Calculator类的Add()和Substract()方法，对其进行日志记录和性能监测。而实现日志记录和性能监测的逻辑代码，则完全与Calculator类的Add()和Substract()方法分开，实现了两者之间依赖的解除，有利于模块的重用和扩展。

4.4.3.2应用程序（Program）  
我们可以实现简单的应用程序，来看看业务对象Calculator施加了日志方面和性能检测方面的效果：  
      class Program  
      {            
             [STAThread]  
             static void Main(string[] args)  
             {  
                    Calculator cal = new Calculator();  
                    cal.Add(3,5);  
                    cal.Substract(3,5);  
                    Console.ReadLine();  
             }  
      }

程序创建了一个Calculator对象，同时调用了Add()和Substract()方法。由于Calculator对象被施加了日志方面和性能检测方面，因此运行结果会将方法调用的详细信息和调用前后的运行当前时间打印出来，如图4.3所示：

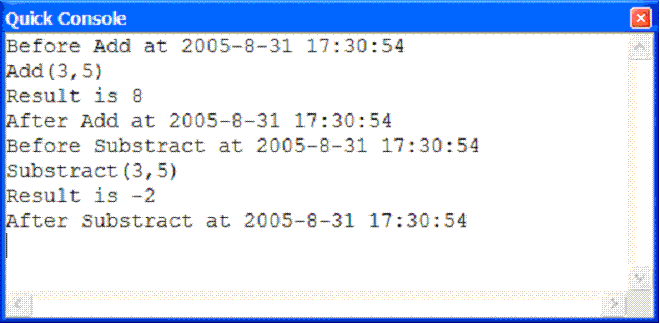


图4.3 施加了方面的业务对象调用结果

如果要改变记录日志和性能监测结果的方式，例如将其写到文件中，则只需要改变LogAdvice和MonitorAdvice的实现，对于Calculator对象而言，则不需要作任何改变。